

msdn

magazine



Developing for
Azure Web Sites

Your Next Great ASP.NET App Starts Here

Deliver interactive touch-enabled user experiences for WebForms and MVC with elegant, high-performance UI controls and extensions from DevExpress.

Download your free 30-day trial at: DevExpress.com/ASP

DX HOTELS Powered by DevExpress ASP.NET Controls and Libraries

Waikiki Beach Hotel ★★★★★

800 Waikiki Beach Rd Honolulu, HI, 96801
Situated directly on Waikiki Beach you are 100 yards away from the fun that awaits you whenever you visit Honolulu. The hotel offers free shuttle service from Honolulu International Airport and we provide you with free towels when using our pool.

Jun 7 - Jun 9, 2014
2 nights, 1 room, 2 adults

What our guests say...

I finally saved up enough money to stay here and I tell ya - this place deserves all the praise it gets.
Wonderful!
Jan 13, 2014
★★★★★

If you've never experienced the Waikiki Beach then I suggest you fly over to Hawaii and see why it's the best.
Feb 15, 2014
★★★★★

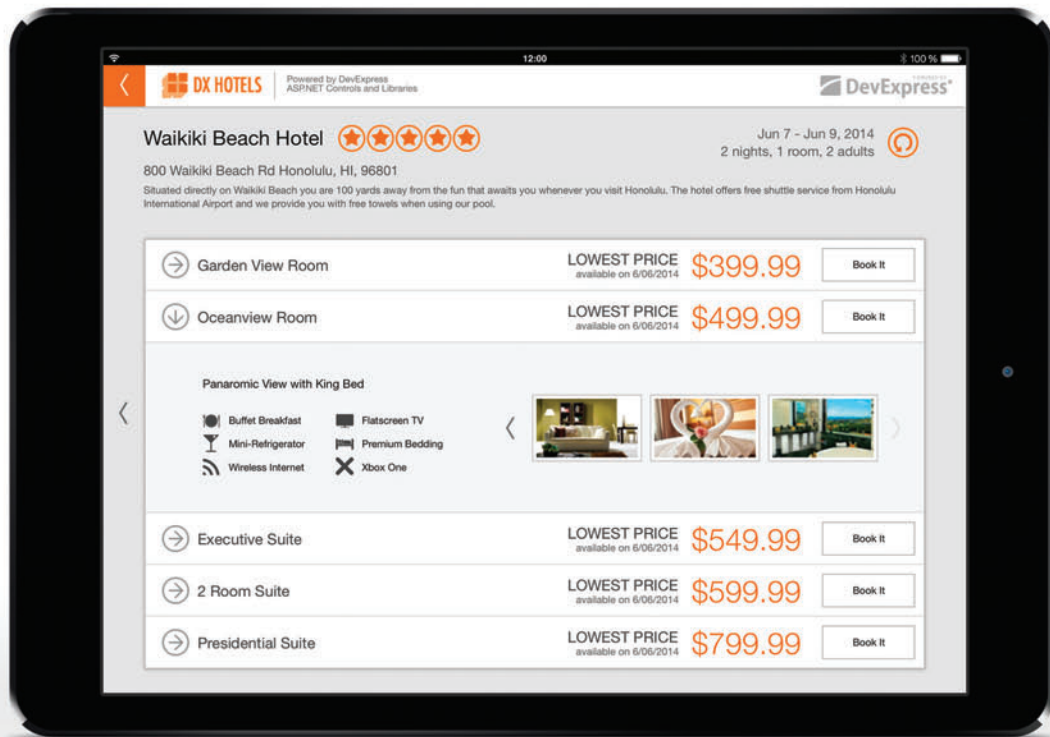
We stayed here on c...
What a hotel! It's ev...
It's totally worth it.
Mar 18, 2014
★★★★★

DevExpress®



Become a UI Superhero

Learn More at DevExpress.com/Superhero



msdn

magazine



Developing for
Azure Web Sites

Scaling Your Web Application with Azure Web Sites Yochay Kiriaty	26
Architecting for the Cloud Using Azure Web Sites Apurva Joshi and Sunitha Muthukrishna	34
Building a Node.js and MongoDB Web Service Tejaswi Redkar	40
Hybrid Connectivity: Connecting Azure Web Sites to LOB Apps Using PortBridge Tejaswi Redkar	48
Teaching from the Cloud James Chambers	56

COLUMNS

WINDOWS WITH C++

Embracing the Windows
Composition Engine
Kenny Kerr, page 8

DATA POINTS

Tips for Updating and
Refactoring Your Entity
Framework Code, Part 2
Julie Lerman, page 16

TEST RUN

Distorting the MNIST
Image Data Set
James McCaffrey, page 66

THE WORKING PROGRAMMER

Fun with C#
Ted Neward, page 70

MODERN APPS

Authentication and Identity
in Windows Store Apps
Rachel Appel, page 72

DIRECTX FACTOR

Breaking the Z Barrier
with Direct2D Effects
Charles Petzold, page 76

DON'T GET ME STARTED

Windows XP:
An Old Soldier Fades Away
David Platt, page 80

#1

**1: As a Scrum team, I want to
get more \$#*% done**

Assigned To: Your Team

Priority: Very High

Release: V 1.0

0 sp  25 sp

INCREASE YOUR TEAM'S VELOCITY AND GET MORE DONE WITH THE #1 SCRUM TOOL.

You can do better than sticky notes. **Axosoft Scrum** is the easiest way to manage backlogs, plan releases and analyze burndown charts. Learn more at [Axosoft.com/MSDNscrum](https://axosoft.com/MSDNscrum).



\$1



A BUG TRACKER FOR YOUR WHOLE TEAM NOW COSTS THE SAME AS A DOLLAR MENU BURGER.

Axosoft Bug Tracker is now just \$1 per year for your entire team. Not \$1 per user, but \$1 for everyone. Check it out at [Axosoft.com/MSDNbugs](https://axosoft.com/MSDNbugs).





dtSearch®

Instantly Search Terabytes of Text

25+ fielded and full-text search types

dtSearch's **own document filters** support "Office," PDF, HTML, XML, ZIP, emails (with nested attachments), and many other file types

Supports databases as well as static and dynamic websites

Highlights hits in all of the above

APIs for .NET, Java, C++, SQL, etc.

64-bit and 32-bit; Win and Linux

"lightning fast" Redmond Magazine

"covers all data sources" eWeek

"results in less than a second" InfoWorld

hundreds more reviews and developer case studies at www.dtsearch.com

dtSearch products:

Desktop with Spider	Web with Spider
Network with Spider	Engine for Win & .NET
Publish (portable media)	Engine for Linux
Document filters also available for separate licensing	

Ask about fully-functional evaluations

The Smart Choice for Text Retrieval® since 1991
www.dtSearch.com 1-800-IT-FINDS

msdn

magazine

JULY 2014 VOLUME 29 NUMBER 7

MOHAMMAD AL-SABT Editorial Director/mmeditor@microsoft.com

KENT SHARKEY Site Manager

MICHAEL DESMOND Editor in Chief/mmeditor@microsoft.com

LAKE LOW Features Editor

SHARON TERDEMAN Features Editor

DAVID RAMEL Technical Editor

WENDY HERNANDEZ Group Managing Editor

SCOTT SHULTZ Creative Director

JOSHUA GOULD Art Director

SENIOR CONTRIBUTING EDITOR Dr. James McCaffrey

CONTRIBUTING EDITORS Rachel Appel, Dino Esposito, Kenny Kerr, Julie Lerman, Ted Neward, Charles Petzold, David S. Platt, Bruno Terkaly, Ricardo Villalobos

Redmond Media Group

Henry Allain President, Redmond Media Group

Michele Imgrund Sr. Director of Marketing & Audience Engagement

Tracy Cook Director of Online Marketing

Irene Fincher Audience Development Manager

ADVERTISING SALES: 818-674-3416/dlbianca@1105media.com

Dan LaBianca Chief Revenue Officer

Chris Kourtoglou Regional Sales Manager

Danna Vedder Regional Sales Manager/Microsoft Account Manager

David Seymour Director, Print & Online Production

Anna Lyn Bayaia Production Coordinator/msdnadproduction@1105media.com

1105 MEDIA

Neal Vitale President & Chief Executive Officer

Richard Vitale Senior Vice President & Chief Financial Officer

Michael J. Valenti Executive Vice President

Christopher M. Coates Vice President, Finance & Administration

Erik A. Lindgren Vice President, Information Technology & Application Development

David F. Myers Vice President, Event Operations

Jeffrey S. Klein Chairman of the Board

MSDN Magazine (ISSN 1528-4859) is published monthly by 1105 Media, Inc., 9201 Oakdale Avenue, Ste. 101, Chatsworth, CA 91311. Periodicals postage paid at Chatsworth, CA 91311-9998, and at additional mailing offices. Annual subscription rates payable in US funds are: U.S. \$35.00, International \$60.00. Annual digital subscription rates payable in U.S. funds are: U.S. \$25.00, International \$25.00. Single copies/back issues: U.S. \$10, all others \$12. Send orders with payment to: *MSDN Magazine*, PO. Box 3167, Carol Stream, IL 60132, email MSDNmag@1105service.com or call (847) 763-9560. **POSTMASTER:** Send address changes to *MSDN Magazine*, PO. Box 2166, Skokie, IL 60076. Canada Publications Mail Agreement No: 40612608. Return Undeliverable Canadian Addresses to Circulation Dept. or XPO Returns: PO. Box 201, Richmond Hill, ON L4B 4R5, Canada.

Printed in the U.S.A. Reproductions in whole or part prohibited except by written permission. Mail requests to "Permissions Editor," c/o *MSDN Magazine*, 4 Venture, Suite 150, Irvine, CA 92618.

Legal Disclaimer: The information in this magazine has not undergone any formal testing by 1105 Media, Inc. and is distributed without any warranty expressed or implied. Implementation or use of any information contained herein is the reader's sole responsibility. While the information has been reviewed for accuracy, there is no guarantee that the same or similar results may be achieved in all environments. Technical inaccuracies may result from printing errors and/or new developments in the industry.

Corporate Address: 1105 Media, Inc., 9201 Oakdale Ave., Ste 101, Chatsworth, CA 91311, www.1105media.com

Media Kits: Direct your Media Kit requests to Matt Morollo, VP Publishing, 508-532-1418 (phone), 508-875-6622 (fax), mmorollo@1105media.com

Reprints: For single article reprints (in minimum quantities of 250-500), e-prints, plaques and posters contact: PARS International, Phone: 212-221-9595, E-mail: 1105reprints@parsintl.com, www.magreprints.com/QuickQuote.asp

List Rental: This publication's subscriber list, as well as other lists from 1105 Media, Inc., is available for rental. For more information, please contact our list manager, Jane Long, Merit Direct. Phone: 913-685-1301; E-mail: jljong@meritdirect.com; Web: www.meritdirect.com/1105

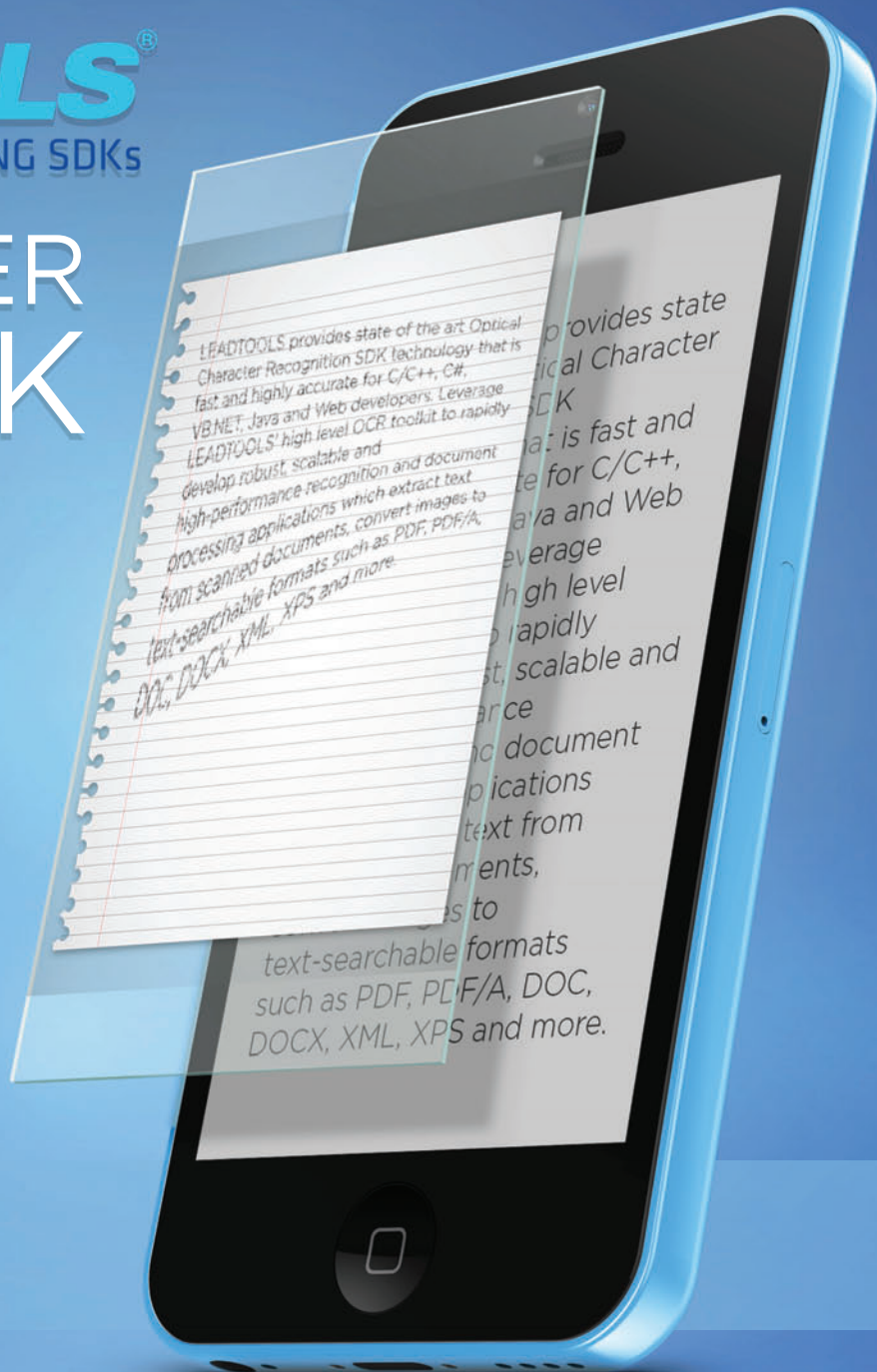
All customer service inquiries should be sent to MSDNmag@1105service.com or call 847-763-9560.



Printed in the USA

THE PREMIER OCR SDK

Converting an
Image to text
has never been
easier. For all
your OCR
needs, look for
LEADTOOLS.



DESKTOP



TABLET



MOBILE

LEADTOOLS provides state of the art Optical Character Recognition SDK technology that is fast and highly accurate for multiple platforms. Leverage LEADTOOLS' high level OCR toolkit to rapidly develop robust, scalable and high-performance recognition and document processing applications which extract text from scanned documents and convert images to text-searchable formats such as PDF, PDF/A, DOC, DOCX, XML, XPS and more.

.NET Windows API WinRT Linux iOS OS X Android HTML5/JavaScript





Inside Azure Web Sites

In case you haven't noticed, Microsoft has been rather busy lately updating and improving its Microsoft Azure cloud platform. I suppose this is what happens when you lure Scott Guthrie over to lead the Azure group at Microsoft (his latest job title: executive vice president of the Cloud and Enterprise group). This is the guy, after all, who helped push the .NET Developer Division into fast-forward, and who helped invent a little thing you might've heard of called ASP.NET.

So it should be no surprise that things are hopping for developers in the Azure division at Microsoft. A spate of recent announcements, including plenty of new features and capabilities unveiled at the Microsoft Build conference in April, illustrate the point. To help developers digest all the new information, this issue of *MSDN Magazine* is focused on Microsoft Azure, and specifically on developing for Azure Web Sites. A package of five feature articles explores how developers can leverage the updated Azure platform to create and deliver powerful, scalable and manageable applications for the Web, quickly and efficiently.

The features lead off with Yochay Kiriati's "Scaling Your Web Application with Azure Web Sites," which shows how to modify Web applications to run across multiple instances and geographies. Next, Apurva Joshi and Sunitha Muthukrishna offer lessons in resilience, as they guide you through optimizing Web applications to weather the challenges of the sometimes-hostile cloud environment.

But wait, there's more. "Building a Node.js and MongoDB Web Service" and "Hybrid Connectivity: Connecting Azure Web Sites to LOB Apps Using PortBridge," both by Tejaswi Redkar, offer wonderful insight into some intriguing use cases. The first article shows how to build a Node.js Web site that draws data from an Azure-hosted MongoDB database, while the second describes how to link and leverage cloud-based Azure Web Sites and on-premises LOB applications. James Chambers closes out the action with his feature, "Teaching from the Cloud," which offers a great content-heavy development scenario, in this case developing an e-learning Azure Web Site.

A key figure in all this is Erez Benari, program manager for IIS and Microsoft Azure Web Sites and the guy who helped us pull together this month's special issue focused on Azure Web Sites. I recently caught up with Benari and asked about the advancements in Azure Web Sites. He talked about new features, such as traffic manager integration, backup and restore, and support for multiple deployment slots, as well as the expansion of Azure into new regions. The changes, Benari says, make Azure Web Sites a "strong offering for large-scale deployments and enterprise customers."

One intriguing aspect of developing for Azure and Azure Web Sites is the immediacy of the experience, which in a sense is changing the relationship between developers and their code. These applications aren't bundled up and thrown over a wall for testing, deployment and use. They're tested in situ, deployed in the blink of an eye, and updated and revised based on real-time telemetry.

Benari says the rapid model demands that developers think like testers and see the big picture. They must go beyond wondering if their code will work and figure out how to quickly fix code with minimal downtime.

"This kind of thinking calls for implementing live-diagnostic information in the code to allow the developer to obtain diagnostic and forensic input from the application as it's being deployed and tested, as well as apply techniques like A/B testing that allow the developer to gather comparative data in real time," Benari says. "Ultimately, it's a matter of responsibility, and as developers get used to being fully accountable (which could sometimes mean a 2:00 a.m. phone call to fix a bug), the quality of software across the board improves, making for a better world for our customers."

Are you on board for this new world? Let me know your thoughts on developing for Microsoft Azure and Azure Web Sites. E-mail me at mmeditor@microsoft.com.

Visit us at msdn.microsoft.com/magazine. Questions, comments or suggestions for *MSDN Magazine*? Send them to the editor: mmeditor@microsoft.com.

© 2014 Microsoft Corporation. All rights reserved.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, you are not permitted to reproduce, store, or introduce into a retrieval system *MSDN Magazine* or any part of *MSDN Magazine*. If you have purchased or have otherwise properly acquired a copy of *MSDN Magazine* in paper format, you are permitted to physically transfer this paper copy in unmodified form. Otherwise, you are not permitted to transmit copies of *MSDN Magazine* (or any part of *MSDN Magazine*) in any form or by any means without the express written permission of Microsoft Corporation.

A listing of Microsoft Corporation trademarks can be found at microsoft.com/library/toolbar/3.0/trademarks/en-us.mspx. Other trademarks or trade names mentioned herein are the property of their respective owners.

MSDN Magazine is published by 1105 Media, Inc. 1105 Media, Inc. is an independent company not affiliated with Microsoft Corporation. Microsoft Corporation is solely responsible for the editorial contents of this magazine. The recommendations and technical guidelines in *MSDN Magazine* are based on specific environments and configurations. These recommendations or guidelines may not apply to dissimilar configurations. Microsoft Corporation does not make any representation or warranty, express or implied, with respect to any code or other information herein and disclaims any liability whatsoever for any use of such code or other information. *MSDN Magazine*, *MSDN*, and Microsoft logos are used by 1105 Media, Inc. under license from owner.

Data Quality Tools for Developers

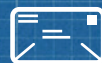
A better way to build in data verification

Since 1985, Melissa Data has provided the tools developers need to enhance databases with clean, correct, and current contact data. Our powerful, yet affordable APIs and Cloud services provide maximum flexibility and ease of integration across industry-standard technologies, including .NET, Java, C, and C++. Build in a solid framework for data quality and protect your investments in data warehousing, business intelligence, and CRM.

- Verify international addresses for over 240 countries
- Enhance contact data with phone numbers and geocodes
- Find, match, and eliminate duplicate records
- Sample source code for rapid application development
- Free trials with 120-day ROI guarantee

Melissa Data.

Architecting data quality success.



Address
Verification



Phone
Verification



Email
Verification



Geocoding



Matching/
Dedupe



Change of
Address

MELISSA DATA®

www.MelissaData.com 1-800-MELISSA

Working with Files?

Convert Print Create Combine Modify



Aspose.Words

DOC, DOCX, RTF, HTML, PDF, XPS & other document formats.



Aspose.Pdf

PDF, XML, XLS-FO, HTML, BMP, JPG, PNG & other image formats.



Aspose.Cells

XLS, XLSX, XLSM, XLTX, CSV, SpreadsheetML & image formats.



Aspose.Slides

PPT, PPTX, POT, POTX, XPS, HTML, PNG, PDF & other formats.



Aspose.Email

MSG, EML, PST, EMLX & other formats.



Aspose.BarCode

JPG, PNG, BMP, GIF, TIFF, WMF, ICON & other image formats.



Aspose.Imaging

PDF, BMP, JPG, GIF, TIFF, PNG, PSD & other image formats.



Aspose.Tasks

XML, MPP, SVG, PDF, TIFF, PNG, CSV, MPT & other formats.



Aspose.Diagram

VSD, VSDX, VSS, VST, VSX & other formats.



Aspose.Note

ONE, PNG, JPG, BMP, GIF & PDF

... and more!

100% Standalone - No Office Automation



.NET Libraries



Java Libraries



Cloud APIs



Android Libraries



Scan for a 20% saving!



Collaborating with Files?

View Sign Annotate Assemble Compare



GroupDocs.Viewer

Native-text, high-fidelity HTML5 document viewer with support for over 45 file formats.



GroupDocs.Signature

Electronic signature API that gives your apps legally binding e-signature capabilities.



GroupDocs.Conversion

Universal document converter for fast conversion between more than 45 file formats.



GroupDocs.Annotation

A powerful API that lets developers annotate Microsoft Office, PDF and other documents within their own apps.



GroupDocs.Assembly

Incorporates data entered by users through online forms Into both Microsoft Office and PDF documents.



GroupDocs.Comparison

A diff view API that allows end users to quickly find differences between two revisions of a document.

100% Standalone - No Office Automation



.NET Libraries



Java Libraries



Cloud APIs



Cloud Apps

document collaboration APIs



GROUPDOCS
Your Document Collaboration APIs

SALES INQUIRIES: +1 214 329 9760 sales@groupdocs.com www.groupdocs.com



Embracing the Windows Composition Engine

The Windows composition engine represents a departure from a world in which each DirectX app requires its own swap chain to one where even so fundamental a construct is unnecessary. Sure, you can continue to write Direct3D and Direct2D apps using a swap chain for presentation, but you no longer have to. The composition engine brings us that much closer to the metal—the GPU—by allowing apps to create composition surfaces directly.

The composition engine's only objective is to compose different bitmaps together. You might request various effects, transforms and even animations to be produced, but at the end of the day it's about composing bitmaps. The engine itself has no graphics-rendering capabilities such as those provided by Direct2D or Direct3D, and it doesn't define vectors or text. What it cares about is composition. Give it a collection of bitmaps and it will do amazing things to combine and compose them together.

Those bitmaps can take any number of forms. A bitmap could actually be video memory. It could even be a swap chain, as I illustrated in my June column (msdn.microsoft.com/magazine/dn745861). But if you really want to begin embracing the composition engine, you need to take a look at composition surfaces. A composition surface is a bitmap provided directly by the composition engine and, as such, allows for certain optimizations that would be difficult to achieve with other forms of bitmaps.

This month, I'm going to take the alpha-blended window from my previous column and show you how it can be reproduced using a composition surface rather than a swap chain. There are some interesting benefits—and also some unique challenges, particularly around handling device loss and per-monitor DPI scaling. But first I need to revisit the issue of window management.

In my previous columns I've either used ATL for window management or illustrated how to register, create and pump window messages directly with the Windows API. There are pros and cons to both approaches. ATL continues to work just fine for window management, but it's slowly losing developer mindshare and Microsoft certainly stopped investing in it long ago. On the other hand, creating a window directly with RegisterClass and CreateWindow tends to be problematic because you can't easily associate a C++ object with a window handle. If you've ever thought of arranging such a union, you may have been tempted to take a peek at the ATL source code to see how this could be achieved, only to realize that there's some dark magic to it with things called "thunks" and even assembly language.

The good news is that it needn't be that hard. Though ATL certainly produces very efficient message dispatching, a simple

Figure 1 A Simple Window Class Template

```
template <typename T>
struct Window
{
    HWND m_window = nullptr;

    static T * GetThisFromHandle(HWND window)
    {
        return reinterpret_cast<T *>(GetWindowLongPtr(window,
                                                    GWLP_USERDATA));
    }

    static LRESULT __stdcall WndProc(HWND    const window,
                                     UINT    const message,
                                     WPARAM  const wparam,
                                     LPARAM  const lparam)
    {
        ASSERT(window);

        if (WM_NCCREATE == message)
        {
            CREATESTRUCT * cs = reinterpret_cast<CREATESTRUCT *>(lparam);
            T * that = static_cast<T *>(cs->lpCreateParams);

            ASSERT(that);
            ASSERT(!that->m_window);

            that->m_window = window;

            SetWindowLongPtr(window,
                            GWLP_USERDATA,
                            reinterpret_cast<LONG_PTR*>(that));
        }
        else if (T * that = GetThisFromHandle(window))
        {
            return that->MessageHandler(message,
                                         wparam,
                                         lparam);
        }

        return DefWindowProc(window,
                              message,
                              wparam,
                              lparam);
    }

    LRESULT MessageHandler(UINT    const message,
                           WPARAM  const wparam,
                           LPARAM  const lparam)
    {
        if (WM_DESTROY == message)
        {
            PostQuitMessage(0);
            return 0;
        }

        return DefWindowProc(m_window,
                              message,
                              wparam,
                              lparam);
    }
};
```


solution involving only standard C++ will do the trick just fine. I don't want to get too sidetracked with the mechanics of window procedures, so I'll simply direct you to **Figure 1**, which provides a simple class template that makes the necessary arrangements to associate a "this" pointer with a window. The template uses the WM_NCCREATE message to extract the pointer and stores it with the window handle. It subsequently retrieves the pointer and sends messages to the most derived message handler.

The assumption is that a derived class will create a window and pass this pointer as the last parameter when calling the CreateWindow or CreateWindowEx functions. The derived class can simply register and create the window, and then respond to window messages with a MessageHandler override. This override relies on compile-time polymorphism so there's no need for virtual functions. However, the effect is the same, so you still need to worry about reentrancy. **Figure 2** shows a concrete window class that relies on the Window class template. This class registers and creates the window in its constructor, but it relies on the window procedure provided by its base class.

Notice that in the constructor in **Figure 2**, the m_window inherited member is uninitialized (a nullptr) prior to calling CreateWindow, but initialized when this function returns. This may seem like magic but it's the window procedure that hooks this up as messages begin arriving, long before CreateWindow returns. The reason it's important to keep this in mind is that by using code such as this you can reproduce the same dangerous effect as calling virtual functions from a constructor. If you end up deriving further, just be sure to pull the window creation out of the constructor so that this form of reentrancy doesn't trip you up. Here's a simple WinMain function that can create the window and pump the window messages:

```
int __stdcall wWinMain(HINSTANCE, HINSTANCE, PWSTR, int)
{
    SampleWindow window;
    MSG message;

    while (GetMessage(&message, nullptr, 0, 0))
    {
        DispatchMessage(&message);
    }
}
```

OK, back to the topic at hand. Now that I have a simple window class abstraction, I can use it to more easily manage the collection of resources needed to build a DirectX application. I'm also going to show you how to properly handle DPI scaling. While I covered DPI scaling in detail in my February 2014 column (msdn.microsoft.com/magazine/dn574798), there are some unique challenges when you combine it with the DirectComposition API. I'll start at the top. I need to include the shell scaling API:

```
#include <ShellScalingAPI.h>
#pragma comment(lib, "shcore")
```

I can now begin to assemble the resources I need to bring my window to life. Given that I have a window class, I can simply make these members of the class. First, the Direct3D device:

```
ComPtr<ID3D11Device> m_device3d;
```

Next, the DirectComposition device:

```
ComPtr<IDCompositionDesktopDevice> m_device;
```

In my previous column I used the IDCompositionDevice interface to represent the composition device. That's the interface

that originated in Windows 8, but it has since been replaced in Windows 8.1 with the IDCompositionDesktopDevice interface, which derives from another new interface called IDCompositionDevice2. These are not related to the original. The IDCompositionDevice2 interface serves to create most of the composition resources and also controls transactional composition. The IDCompositionDesktopDevice interface adds the ability to create certain window-specific composition resources.

I'll also need a composition target, visual and surface:

```
ComPtr<IDCompositionTarget> m_target;
ComPtr<IDCompositionVisual2> m_visual;
ComPtr<IDCompositionSurface> m_surface;
```

The composition target represents the binding between the desktop window and a visual tree. I can actually associate two visual trees with a given window, but more on that in a future column. The visual represents a node in the visual tree. I'm going to explore visuals in a subsequent column, so for now there'll be just a single root visual. Here, I'm just using the IDCompositionVisual2 interface, which derives from the IDCompositionVisual interface I used in my previous column. Finally, there's the surface

Figure 2 A Concrete Window Class

```
struct SampleWindow : Window<SampleWindow>
{
    SampleWindow()
    {
        WNDCLASS wc = {};

        wc.hCursor      = LoadCursor(nullptr, IDC_ARROW);
        wc.hInstance    = reinterpret_cast<HINSTANCE>(&__ImageBase);
        wc.lpszClassName = L"SampleWindow";
        wc.style         = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc   = WndProc;

        RegisterClass(&wc);

        ASSERT(!m_window);

        VERIFY(CreateWindowEx(WS_EX_NOREDIRECTIONBITMAP,
                               wc.lpszClassName,
                               L"Window Title",
                               WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                               CW_USEDEFAULT, CW_USEDEFAULT,
                               CW_USEDEFAULT, CW_USEDEFAULT,
                               nullptr,
                               nullptr,
                               wc.hInstance,
                               this));

        ASSERT(m_window);
    }

    LRESULT MessageHandler(UINT message,
                           WPARAM const wparam,
                           LPARAM const lparam)
    {
        if (WM_PAINT == message)
        {
            PaintHandler();
            return 0;
        }

        return __super::MessageHandler(message,
                                         wparam,
                                         lparam);
    }

    void PaintHandler()
    {
        // Render ...
    }
};
```

representing the content or bitmap associated with the visual. In my previous column I simply used a swap chain as the content of a visual, but in a moment I'm going to show you how to create a composition surface instead.

To illustrate how to actually render something and manage the rendering resources, I need a few more member variables:

```
ComPtr<ID2D1SolidColorBrush> m_brush;
D2D_SIZE_F m_size;
D2D_POINT_2F m_dpi;

SampleWindow() :
    m_size(),
    m_dpi()
{
    // RegisterClass / CreateWindowEx as before
}
```

The Direct2D solid color brush is quite inexpensive to create, but many other rendering resources are not so lightweight. I'll use this brush to illustrate how to create rendering resources outside of the rendering loop. The DirectComposition API also optionally takes over the creation of the Direct2D render target. This allows you to target a composition surface with Direct2D, but it also means you lose a bit of contextual information. Specifically, you can no longer cache the applicable DPI scaling factor in the render target because DirectComposition creates it for you on demand. Also, you can no longer rely on the render target's `GetSize` method to report the size of the window. But don't worry, I'll show you how to make up for these drawbacks in a moment.

As with any application that relies on a Direct3D device, I need to be careful to manage the resources that reside on the physical device, on the assumption the device may be lost at any moment. The GPU might hang, reset, be removed or simply crash. In addition, I need to be careful not to respond inappropriately to window messages that might arrive before the device stack is created. I'll use the Direct3D device pointer to indicate whether the device has been created:

```
bool IsDeviceCreated() const
{
    return m_device3d;
}
```

This just helps to make the query explicit. I'll also use this pointer to initiate a reset of the device stack to force all the device-dependent resources to be recreated:

```
void ReleaseDeviceResources()
{
    m_device3d.Reset();
}
```

Again, this just helps to make this operation explicit. I could release all of the device-dependent resources here, but that's not strictly necessary and it can quickly become a maintenance headache as different resources are added or removed. The meat of the device creation process lives in another helper method:

```
void CreateDeviceResources()
{
    if (IsDeviceCreated()) return;

    // Create devices and resources ...
}
```

It's here in the `CreateDeviceResources` method that I can create or recreate the device stack, the hardware device, and the various resources the window requires. First, I create the Direct3D device upon which everything else rests:

```
HR(D3D11CreateDevice(nullptr, // Adapter
    D3D_DRIVER_TYPE_HARDWARE,
    nullptr, // Module
    D3D11_CREATE_DEVICE_BGRA_SUPPORT,
    nullptr, 0, // Highest available feature level
    D3D11_SDK_VERSION,
    m_device3d.GetAddressOf(),
    nullptr, // Actual feature level
    nullptr)); // Device context
```

Notice how the resulting interface pointer is captured by the `m_device3d` member. Now, I need to query for the device's DXGI interface:

```
ComPtr<IDXGIDevice> devicex;
HR(m_device3d.As(&devicex));
```

In my previous column, it was at this point that I created the DXGI factory and swap chain to be used for composition. I created a swap chain, wrapped it in a Direct2D bitmap, targeted the bitmap with a device context and so on. Here, I'm going to do things quite differently. Having created the Direct3D device, I'm now going to create a Direct2D device pointing to it, and then create DirectComposition device pointing to the Direct2D device. Here's the Direct2D device:

```
ComPtr<ID2D1Device> device2d;

HR(D2D1CreateDevice(devicex.Get(),
    nullptr, // Default properties
    device2d.GetAddressOf()));
```

I use a helper function provided by the Direct2D API instead of the more familiar Direct2D factory object. The resulting Direct2D device simply inherits the threading model from the DXGI device, but you can override that and turn on debug tracing as well. Here's the DirectComposition device:

```
HR(DCompositionCreateDevice2(
    device2d.Get(),
    __uuidof(m_device),
    reinterpret_cast<void **>(m_device.ReleaseAndGetAddressOf())));
```

I'm careful to use the `m_device` member's `ReleaseAndGetAddressOf` method to support the recreation of the device stack after device loss. And given the composition device, I can now create the composition target as I did in my previous column:

```
HR(m_device->CreateTargetForHwnd(m_window,
    true, // Top most
    m_target.ReleaseAndGetAddressOf()));
```

And the root visual:

```
HR(m_device->CreateVisual(m_visual.ReleaseAndGetAddressOf()));
```

Now it's time to focus on the composition surface that replaces the swap chain. In the same way the DXGI factory has no idea how big a swap chain's buffers ought to be when I call the `CreateSwapChainForComposition` method, the DirectComposition device has no idea how big the underlying surface ought to be. I need to query the size of the window's client area and use that information to inform the creation of the surface:

```
RECT rect = {};

VERIFY(GetClientRect(m_window,
    &rect));
```

The `RECT` struct has left, top, right, and bottom members and I can use those to determine the desired size of the surface to be created using physical pixels:

```
HR(m_device->CreateSurface(rect.right - rect.left,
    rect.bottom - rect.top,
    DXGI_FORMAT_B8G8R8A8_UNORM,
    DXGI_ALPHA_MODE_PREMULTIPLIED,
    m_surface.ReleaseAndGetAddressOf()));
```

Switch to Amyuni PDF

Create & Edit PDFs in .Net - ActiveX - WinRT

- Edit, process and print PDF 1.7 documents
- Create, fill-out and annotate PDF forms
- Fast and lightweight 32- and 64-bit components for .Net and ActiveX/COM
- New WinRT Component enables publishing C#, C++CX or Javascript apps to Windows Store
- New Postscript/EPS to PDF conversion module



Complete Suite of Accurate PDF Components

- All your PDF processing, conversion and editing in a single package
- Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment
- Includes our Microsoft WHQL certified PDF Converter printer driver
- Export PDF documents into other formats such as JPeg, PNG, XAML or HTML5
- Import and Export XPS files using any programming environment



Advanced HTML to PDF & XAML

- Direct conversion of HTML files into PDF and XAML without the use of a web browser or a printer driver
- Easy Integration and deployment within developer's applications
- WebkitPDF is based on the Webkit Open Source library and Amyuni PDF Creator



High Performance PDF Printer For Desktops and Servers



- Our high-performance printer driver optimized for Web, Application and Print Servers. Print to PDF in a fraction of the time needed with other tools. WHQL tested for Windows 32 and 64-bit including Windows Server 2012 R2 and Windows 8.1
- Standard PDF features included with a number of unique features. Interface with any .Net or ActiveX programming language
- Easy licensing and deployment to fit system administrator's requirements



AMYUNI 

All development tools available at

www.amyuni.com

USA and Canada

Toll Free: 1866 926 9864
Support: 514 868 9227
sales@amyuni.com

Europe

UK: 0800-015-4682
Germany: 0800-183-0923
France: 0800-911-248

Keep in mind the actual surface may well be larger than the size requested. This is because the composition engine might pool or round allocations for greater efficiency. This isn't a problem, but it does affect the resulting device context because you won't be able to rely on its `GetSize` method, but more on that in a moment.

The parameters to the `CreateSurface` method are, thankfully, a simplification of the `DXGI_SWAP_CHAIN_DESC1` structure's many knobs and dials. Following the size, I specify the pixel format and alpha mode, and the composition device returns a pointer to the newly created composition surface. I can then simply set this surface as the content of my visual object and set the visual as the root of my composition target:

```
HR(m_visual->SetContent(m_surface.Get()));
HR(m_target->SetRoot(m_visual.Get()));
```

The parameters to the `CreateSurface` method are, thankfully, a simplification of the `DXGI_SWAP_CHAIN_DESC1` structure's many knobs and dials.

I do not, however, need to call the composition device's `Commit` method at this stage. I'm going to be updating the composition surface in my rendering loop, but those changes will take effect only when the `Commit` method is called. At this point, the composition engine is ready for me to begin rendering, but I still have a few loose ends to tie up. They have nothing to do with composition, but everything to do with correctly and efficiently using `Direct2D` for rendering. First, any render target-specific resources such as bitmaps and brushes ought to be created outside of the rendering loop. This can be a little awkward because `DirectComposition` creates the render target. Fortunately, the only requirement is that these resources be created in the same address space as the eventual render target, so I can simply create a throw-away device context here in order to create such resources:

```
ComPtr<ID2D1DeviceContext> dc;
```

```
HR(device2d->CreateDeviceContext(D2D1_DEVICE_CONTEXT_OPTIONS_NONE,
    dc.GetAddressOf()));
```

I can then use this render target to create the app's single brush:

```
D2D_COLOR_F const color = ColorF(0.26f,
    0.56f,
    0.87f,
    0.5f);
```

```
HR(dc->CreateSolidColorBrush(color,
    m_brush.ReleaseAndGetAddressOf()));
```

The device context is then discarded but the brush remains to be reused in my rendering loop. This is somewhat counterintuitive, but it will make sense in a moment. The last thing I need to do before rendering is to fill in those two member variables `m_size` and `m_dpi`. Traditionally, a `Direct2D` render target's `GetSize` method provides the size of the render target in logical pixels, otherwise known as device-independent pixels. This logical size already

Figure 3 Creating the Device Stack

```
void CreateDeviceResources()
{
    if (IsDeviceCreated()) return;

    HR(D3D11CreateDevice(nullptr, // Adapter
        D3D_DRIVER_TYPE_HARDWARE,
        nullptr, // Module
        D3D11_CREATE_DEVICE_BGRA_SUPPORT,
        nullptr, 0, // Highest available feature level
        D3D11_SDK_VERSION,
        m_device3d.GetAddressOf(),
        nullptr, // Actual feature level
        nullptr)); // Device context

    ComPtr<IDXGIDevice> devicecx;
    HR(m_device3d.As(&devicecx));

    ComPtr<ID2D1Device> device2d;

    HR(D2D1CreateDevice(devicecx.Get(),
        nullptr, // Default properties
        device2d.GetAddressOf()));

    HR(DCompositionCreateDevice2(
        device2d.Get(),
        __uuidof(m_device),
        reinterpret_cast<void **>(m_device.ReleaseAndGetAddressOf())));

    HR(m_device->CreateTargetForHwnd(m_window,
        true, // Top most
        m_target.ReleaseAndGetAddressOf()));

    HR(m_device->CreateVisual(m_visual.ReleaseAndGetAddressOf()));

    RECT rect = {};

    VERIFY(GetClientRect(m_window,
        &rect));

    HR(m_device->CreateSurface(rect.right - rect.left,
        rect.bottom - rect.top,
        DXGI_FORMAT_B8G8R8A8_UNORM,
        DXGI_ALPHA_MODE_PREMULTIPLIED,
        m_surface.ReleaseAndGetAddressOf()));

    HR(m_visual->SetContent(m_surface.Get()));
    HR(m_target->SetRoot(m_visual.Get()));

    ComPtr<ID2D1DeviceContext> dc;

    HR(device2d->CreateDeviceContext(D2D1_DEVICE_CONTEXT_OPTIONS_NONE,
        dc.GetAddressOf()));

    D2D_COLOR_F const color = ColorF(0.26f,
        0.56f,
        0.87f,
        0.5f);

    HR(dc->CreateSolidColorBrush(color,
        m_brush.ReleaseAndGetAddressOf()));

    HMONITOR const monitor = MonitorFromWindow(m_window,
        MONITOR_DEFAULTTONEAREST);

    unsigned x = 0;
    unsigned y = 0;

    HR(GetDpiForMonitor(monitor,
        MDT_EFFECTIVE_DPI,
        &x,
        &y));

    m_dpi.x = static_cast<float>(x);
    m_dpi.y = static_cast<float>(y);

    m_size.width = (rect.right - rect.left) * 96 / m_dpi.x;
    m_size.height = (rect.bottom - rect.top) * 96 / m_dpi.y;
}
```




Desktop. Web. Mobile. Your next great app starts here.

From interactive Desktop applications, to immersive Web and Mobile solutions, development tools built to meet your needs today and ensure your continued success tomorrow.

Download your free 30-day trial today and Experience the DevExpress Difference.

www.devexpress.com/try

DevExpress®

WIN ASP WPF SL VCL XAF CR

All trademarks or registered trademarks are property of their respective owners.

accounts for the effective DPI, so I'll deal with that first. As I illustrated in my February 2014 column about high-DPI apps, I can query the effective DPI for a particular window by first determining the monitor on which a given window predominantly resides and then getting the effective DPI for that monitor. Here's what that looks like:

```
HMONITOR const monitor = MonitorFromWindow(m_window,
                                          MONITOR_DEFAULTTONEAREST);

unsigned x = 0;
unsigned y = 0;

HR(GetDpiForMonitor(monitor,
                   MDT_EFFECTIVE_DPI,
                   &x,
                   &y));
```

I can then cache these values in my `m_dpi` member so I can easily update the device context provided by the `DirectComposition` API inside my rendering loop:

```
m_dpi.x = static_cast<float>(x);
m_dpi.y = static_cast<float>(y);
```

Now, calculating the logical size of the client area in logical pixels is a simple matter of taking the `RECT` structure that already holds the size in physical pixels and factoring in the effective DPI values I now have handy:

```
m_size.width = (rect.right - rect.left) * 96 / m_dpi.x;
m_size.height = (rect.bottom - rect.top) * 96 / m_dpi.y;
```

And that concludes the `CreateDeviceResources` method and all it's responsible for. You can see how it all comes together in **Figure 3**, which shows the `CreateDeviceResources` method in its entirety.

Before implementing the message handlers, I need to override the `Window` class template's `MessageHandler` to indicate which messages I'd like to handle. At a minimum, I need to handle the `WM_PAINT` message where I'll provide the drawing commands; the `WM_SIZE` message where I'll adjust the surface size; and the `WM_DPICHANGED` message where I'll update the effective DPI and size of the window. **Figure 4** shows the `MessageHandler` and, as you'd expect, it simply forwards the messages on to the appropriate handlers.

The `WM_PAINT` handler is where I create the device resources on-demand before entering the drawing sequence. Remember that `CreateDeviceResources` does nothing if the device already exists:

```
void PaintHandler()
{
    try
    {
        CreateDeviceResources();

        // Drawing commands ...
    }
}
```

In this way, I can simply react to device loss by releasing the `Direct3D` device pointer through the `ReleaseDeviceResources` method, and the next time around the `WM_PAINT` handler will recreate it all. The entire process is enclosed in a try block so that any device failure can be handled reliably. To begin drawing to the composition surface, I need to call its `BeginDraw` method:

```
ComPtr<ID2D1DeviceContext> dc;
POINT offset = {};

HR(m_surface->BeginDraw(nullptr, // Entire surface
    _uuidof(dc),
    reinterpret_cast<void **>(dc.GetAddressOf()),
    &offset));
```

`BeginDraw` returns a device context—the `Direct2D` render target—that I'll use to batch up the actual drawing commands. The `DirectComposition` API uses the `Direct2D` device I originally

provided when creating the composition device to create and return the device context here. I can optionally provide a `RECT` structure in physical pixels to clip the surface or specify a nullptr to allow unrestricted access to the drawing surface. The `BeginDraw` method also returns an offset, again in physical pixels, to indicate the origin of the intended drawing surface. This will not necessarily be the top-left corner of the surface and care must be taken to adjust or transform any drawing commands so they're properly offset.

The composition surface also sports an `EndDraw` method and these two take the place of the `Direct2D` `BeginDraw` and `EndDraw` methods. You must not call the corresponding methods on the device context as the `DirectComposition` API takes care of this for you. Obviously, the `DirectComposition` API also ensures that the device context has the composition surface selected as its target. Moreover, it's important you don't hold on to the device context but release it promptly after drawing concludes. Also, the surface isn't guaranteed to retain the contents of any previous frame that may have been drawn, so care needs to be taken to either clear the target or redraw every pixel before concluding.

The resulting device context is ready to go but doesn't have the window's effective DPI scaling factor applied. I can use the DPI values I previously calculated inside my `CreateDeviceResources` method to update the device context now:

```
dc->SetDpi(m_dpi.x,
          m_dpi.y);
```

I'll also simply use a translation transformation matrix to adjust the drawing commands given the offset required by the `DirectComposition` API. I just need to be careful to translate the offset to logical pixels because that's what `Direct2D` assumes:

```
dc->SetTransform(Matrix3x2F::Translation(offset.x * 96 / m_dpi.x,
                                          offset.y * 96 / m_dpi.y));
```

I can now clear the target and draw something app-specific. Here, I draw a simple rectangle with the device-dependent brush I created earlier in my `CreateDeviceResources` method:

```
dc->Clear();

D2D_RECT_F const rect = RectF(100.0f,
                              100.0f,
                              m_size.width - 100.0f,
                              m_size.height - 100.0f);

dc->DrawRectangle(rect,
                 m_brush.Get(),
                 50.0f);
```

I'm relying on the cached `m_size` member rather than any size reported by the `GetSize` method, as the latter reports the size of the underlying surface rather than the client area.

Concluding the drawing sequence involves a number of steps. First, I need to call the `EndDraw` method on the surface. This tells `Direct2D` to complete any batched drawing commands and write them to the composition surface. The surface is then ready to be composed—but not before the `Commit` method is called on the composition device. At that point, any changes to the visual tree, including any updated surfaces, are batched up and made available to the composition engine in a single transactional unit. This concludes the rendering process. The only remaining question is whether the `Direct3D` device has been lost. The `Commit` method will report any failure and the catch block will release the device. If all goes well, I can tell Windows that I've successfully painted

Figure 4 Dispatching Messages

```
LRESULT MessageHandler(UINT message,
                        WPARAM const wparam,
                        LPARAM const lparam)
{
    if (WM_PAINT == message)
    {
        PaintHandler();
        return 0;
    }

    if (WM_SIZE == message)
    {
        SizeHandler(wparam, lparam);
        return 0;
    }

    if (WM_DPICHANGED == message)
    {
        DpiHandler(wparam, lparam);
        return 0;
    }

    return __super::MessageHandler(message,
                                    wparam,
                                    lparam);
}
```

the window by validating the window's entire client area with the `ValidateRect` function. Otherwise, I need to release the device. Here's what this might look like:

```
// Drawing commands ...

HR(m_surface->EndDraw());
HR(m_device->Commit());

VERIFY(ValidateRect(m_window, nullptr));
}
catch (ComException const & e)
{
    ReleaseDeviceResources();
}
```

I don't need to repaint explicitly, because Windows will simply continue to send `WM_PAINT` messages if I don't respond by validating the client area. The `WM_SIZE` handler is responsible for adjusting the size of the composition surface and also for updating the cached size of the render target. I won't react if the device isn't created or the window is minimized:

```
void SizeHandler(WPARAM const wparam,
                LPARAM const lparam)
{
    try
    {
        if (!IsDeviceCreated()) return;
        if (SIZE_MINIMIZED == wparam) return;

        // ...
    }
}
```

A window typically receives a `WM_SIZE` message before it's had an opportunity to create the device stack. When that happens, I simply ignore the message. I also ignore the message if the `WM_SIZE` message is a result of a minimized window. I don't want to unnecessarily adjust the size of the surface in that case. As with the `WM_PAINT` handler, the `WM_SIZE` handler encloses its operations in a try block. Resizing, or in this case recreating, the surface may well fail due to device loss and this ought to result in the device stack being recreated. But, first, I can extract the new size of the client area:

```
unsigned const width  = LOWORD(lparam);
unsigned const height = HIWORD(lparam);
```

Figure 5 Handling DPI Updates

```
void DpiHandler(WPARAM const wparam,
               LPARAM const lparam)
{
    m_dpi.x = LOWORD(wparam);
    m_dpi.y = HIWORD(wparam);

    RECT const & rect = *reinterpret_cast<RECT const *>(lparam);

    VERIFY(SetWindowPos(m_window,
                        0, // No relative window
                        rect.left,
                        rect.top,
                        rect.right - rect.left,
                        rect.bottom - rect.top,
                        SWP_NOACTIVATE | SWP_NOZORDER));
}
```

And update the cached size in logical pixels:

```
m_size.width  = width * 96 / m_dpi.x;
m_size.height = height * 96 / m_dpi.y;
```

The composition surface isn't resizable. I'm using what might be called a non-virtual surface. The composition engine also offers virtual surfaces that are resizable but I'll talk more about that in an upcoming column. Here, I can simply release the current surface and recreate it. Because changes to the visual tree aren't reflected until changes are committed, the user won't experience any flickering while the surface is discarded and recreated. Here's what this might look like:

```
HR(m_device->CreateSurface(width,
                          height,
                          DXGI_FORMAT_B8G8R8A8_UNORM,
                          DXGI_ALPHA_MODE_PREMULTIPLIED,
                          m_surface.ReleaseAndGetAddressOf());

HR(m_visual->SetContent(m_surface.Get()));
```

I can then respond to any failure by releasing the device resources so that the next `WM_PAINT` message will cause them to be recreated:

```
// ...
}
catch (ComException const & e)
{
    ReleaseDeviceResources();
}
```

That's it for the `WM_SIZE` handler. The final necessary step is to implement the `WM_DPICHANGED` handler to update the effective DPI and size of the window. The message's `WPARAM` provides the new DPI values and the `LPARAM` provides the new size. I simply update the window's `m_dpi` member variable and then call the `SetWindowPos` method to update the window's size. The window will then receive another `WM_SIZE` message that my `WM_SIZE` handler will use to adjust the `m_size` member and recreate the surface. **Figure 5** provides an example of how to handle these `WM_DPICHANGED` messages.

I'm happy to see the members of the DirectX family drawing closer together with improved interoperability and performance, thanks to deep integration between `Direct2D` and `DirectComposition`. I hope you're as excited as I am about the possibilities for building rich native apps with DirectX. ■

KENNY KERR is a computer programmer based in Canada, as well as an author for *Pluralsight* and a Microsoft MVP. He blogs at kennykerr.ca and you can follow him on Twitter at twitter.com/kennykerr.

THANKS to the following Microsoft technical experts for reviewing this article: *Leonardo Blanco and James Clarke*



Tips for Updating and Refactoring Your Entity Framework Code, Part 2

Entity Framework (EF) has been around for more than eight years and has gone through a lot of changes. This means developers are looking at applications that use early versions of EF to see how they can benefit from its newer enhancements. I've worked with a number of software teams going through this process and am sharing some lessons learned and guidance in a two-part series. Last month I discussed updating to EF6, using an old EF4 application as an example (bit.ly/1jqu5yQ). I also provided tips for breaking out smaller Entity Data Models from large models (EDMX or Code First), because I recommend working with multiple focused models rather than using one large model—laden with the complication of often unnecessary relationships—throughout your entire application.

This month, I'll use a specific example to share more details about breaking out a small model, and then, using that small model, work through some of the problems you encounter when switching from theObjectContext API to the newer DbContext API.

For this article, I use an existing app that's a bit more than the typical demo app in order to present the kind of real-world problems you might run into when refactoring your own solutions. It's a small sample application from my book, "Programming Entity Framework, 2nd Edition" (O'Reilly Media, 2010), written using EF4 and the ObjectContext API. Its EDMX model was generated using Database First, then customized in the EF Designer. In this solution, I had shifted away from the default T4 code generation template that creates entity classes that all inherit from the EntityObject class. Instead, I used a T4 template that created plain old CLR objects (POCOs), which were supported beginning with EF4, and I had customized the template a bit. I should note that this is a client-side application and, therefore, doesn't have some of the challenges of a disconnected app, where state tracking is more challenging. Regardless, many of the issues you'll see here apply to both scenarios.

In last month's column, I upgraded this solution to use EF6 and the Microsoft .NET Framework 4.5, but I didn't make any changes to my codebase. It's still using the original T4 template to generate POCO classes and an ObjectContext class that manages the persistence.

This is a client-side application and, therefore, doesn't have some of the challenges of a disconnected app.

The front end of the application uses Windows Presentation Foundation (WPF) for its UI. While the architecture is layered, my ideas on architecting applications have definitely evolved since then. Nevertheless, I'll refrain from the full-blown refactor that would make my heart zing when I looked at the updated code.

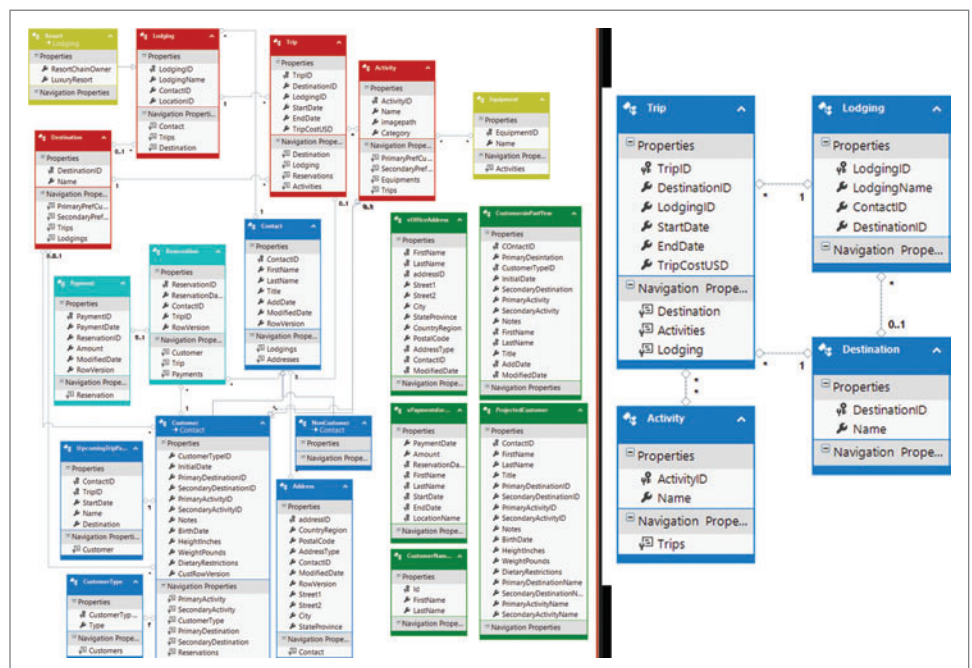


Figure 1 The Full Model on the Left, the Constrained Model on the Right, Created Using the Entities Shown in Red in the Bigger Model

Destination	Date
New Zealand	11/13/2008
New Zealand	11/13/2011
Norway	05/14/2010
Peru	04/08/2014
Peru	04/10/2009
Peru	04/10/2012
Peru	04/07/2009
Peru	04/07/2012
Peru	05/04/2009
Peru	05/04/2012

Start Date: 4/8/2014 End Date: 4/18/2014

Destination: Peru Lodging: Hilton Peru

Activities on this Trip: Kayaking, Walking Tours, Backpacking, Llama Trekking

Buttons: New Trip, Save, Add Activity to Trip

Figure 2 The Windows Presentation Foundation Form for Managing Trip Details

My goals for this column are to:

- Extract a smaller model that's focused on one task.
- Change the code generation of the smaller model to output the newer-style POCO classes and a DbContext class to manage the persistence.
- Fix any existing code that uses theObjectContext and broke because of the switch to DbContext. In many cases, this means creating duplicate methods and then modifying those to DbContext logic. This way, I won't break the remaining code that's still being used by the original model and itsObjectContext.
- Look for opportunities to replace logic with simpler and more efficient functionality introduced in EF5 and EF6.

Creating the Trip Maintenance Model

In last month's column, I provided pointers for identifying and extracting a smaller model from a large one. Following that guidance, I examined the model of this app, which isn't very big, but does contain entities for managing a variety of tasks involved in running an adventure travel business. It has entities for maintaining trips defined by destinations, activities, accommodations, dates, and other details, as well as customers, their reservations, payments, and sundry contact information. You can see the full model on the left side of **Figure 1**.

One of the application's tasks allows users to define new trips as well as to maintain existing trips using a WPF form, as shown in **Figure 2**.

What the WPF form handles is the definition of trips: selecting a destination and a hotel, the start and end dates, and a variety of activities. Therefore, I can envision a model limited to satisfying just this

set of tasks. I created a new model (also shown in **Figure 1**) using the Entity Data Model wizard and selected only the relevant tables. The model is also aware of the join table responsible for the many-to-many relationship between events and activities.

Next, I applied the same customizations (on which my code depends) to these entities in my new model that I had previously defined in the original model. Most of these were name changes to entities and their properties; for example, Event became Trip and Location became Destination.

I'm working with an EDMX, but you can define a new model for Code First by creating a new DbContext class that targets only the four entities. In that case, however, you need to either define new types that don't include the superfluous relationships or use the Fluent API mappings to ignore particular relationships. My January 2013 Data Points column, "Shrink Models with DDD Bounded Contexts" (bit.ly/1isloGE), provides guidance for the Code First path.

It's nice to do away with the challenges of maintaining relationships about which, in this context, I just don't care. For example, reservations, and all of the things tied to reservations (such as payments and customers), are now gone.

I also trimmed Lodging and Activity, because I really need their identity and name only for trip maintenance. Unfortunately, some rules around mapping to non-nullable database columns meant I had to retain CustomerID and DestinationID. But I no longer need navigation properties from Destination or Lodging back to Trip, so I deleted them from the model.

It's nice to do away with the challenges of maintaining relationships about which, in this context, I just don't care.

Next, I had to consider what to do with the code-generated classes. I already have classes generated from the other model, but those classes relate to relationships I don't need (or have) in this model. Because I've been focused on Domain-Driven Design (DDD) for a number of years, I'm comfortable having a set of classes specific to this new model and using them separately from the other generated classes. These classes are in a separate project and a different namespace. Because they map back to a common database, I don't have to worry about changes made in one place not showing up in another. Having classes that are focused just on the task of maintaining trips will make coding simpler, though it will mean some duplication across my solution. The redundancy is a trade-off I'm willing to make and one that I've come to terms with already in previous projects. The left side of **Figure 3** shows the template (BreakAway.tt) and generated classes associated with the large model. They're in their own project, BreakAwayEntities. I'll explain the right side of **Figure 3** shortly.

As per last month's column, when I created the new model, I used my original code-generation template so that my only challenge during

this step would be to make sure my application functions using the small model, without having to concurrently worry about changed EF APIs. I was able to do this by replacing the code inside the default template files (TripMaintenance.Context.tt and TripMaintenance.tt) with the code inside of my BreakAwayEntities.tt files. Additionally, I had to modify the file path in the template code to point to the new EDMX file.

Now I have a smaller model that's much less complicated and will make the task of interacting with this data in my WPF code simpler.

In all, it took me about an hour of changing references and namespaces until I was able to run my little application and my tests again using the new model—not just retrieving data but editing and inserting new graphs of data.

Pulling the Plug: Moving to the DbContext API

Now I have a smaller model that's much less complicated and will make the task of interacting with this data in my WPF code simpler. I'm ready to attack the harder chore: replacing myObjectContext with the DbContext so I can delete code I wrote to make up for the complexity of working directly with the ObjectContext. I'm grateful I'll be tangling only with the smaller surface area of code that's related to my new model. It's like re-breaking a broken arm so it will set properly, while the rest of the bones of my application will remain intact and pain-free.

Updating the Code Generation Template for my EDMX I'll continue to use the EDMX, so in order to switch Trip Maintenance to the DbContext API, I must select a new code-generation template. I can easily access the one I want: EF 6.x DbContext Generator, because it was installed with Visual Studio 2013. First, I'll delete the two TT files that are attached to the TripMaintenance.EDMX file, and then I can use the designer's Add Code Generation Item tool to select the new template. (If you're unfamiliar with using the code-generation templates, see the MSDN document, "EF Designer Code Generation Templates," at bit.ly/1i7zU3Y).

Recall that I had customized the original template. A critical customization I'll need to duplicate in the new template is removing the code that injects the virtual keyword on navigation properties. This will ensure lazy loading doesn't get triggered, as my code depends on this behavior. If you're using a T4 template and have customized anything for your original model, this is a very important step to keep in mind. (You can view an old video I created for MSDN that demonstrates editing an EDMX T4 template at bit.ly/1jKg4jB.)

One last detail was to attend to some partial classes I'd created for some of the entities and the ObjectContext. I copied the relevant ones into the new model project and made sure they were tied to the newly generated classes.

Fixing the AddObject, AttachObject and DeleteObject Methods Now it's time to see the damage. While much of my broken code is particular to how I coded against the ObjectContext API, there's a set of easily targeted methods that will be common to most applications, so I'll hit that first.

The ObjectContext API offers multiple ways to add, attach and remove objects from an ObjectSet. For example, to add an item, such as an instance of trip named newTrip, you could use ObjectContext directly:

```
context.AddObject("Trips",newTrip)
```

Or you could do it from ObjectSet:

```
_context.Trips.AddObject(newTrip)
```

The ObjectContext method is a little clunky because it requires a string to identify to which set the entity belongs. The ObjectSet method is simpler because the set is already defined, but it still uses clunky terms: AddObject, AttachObject and DeleteObject. With DbContext, there's just one way—through DbSet. The method names were simplified to Add, Attach and Remove to better emulate collection methods, for example:

```
_context.Trips.Add(newTrip);
```

Notice that DeleteObject became Remove. This can be confusing because while Remove aligns better with collections, it doesn't really state the intent of the method, which is to eventually delete the record from the database. I've seen developers mistakenly assume that Collection.Remove will have the same result as DbSet.Remove—indicating to EF that the target entity be deleted from the database. But it won't. So do be careful with that.

The rest of the problems I tackle are specific to how I used the ObjectContext in my original application. They aren't necessarily the same problems you'll encounter, but seeing these particular disruptions—and how to fix them—will help you prepare for whatever you encounter when making the switch to DbContext in your own solutions.

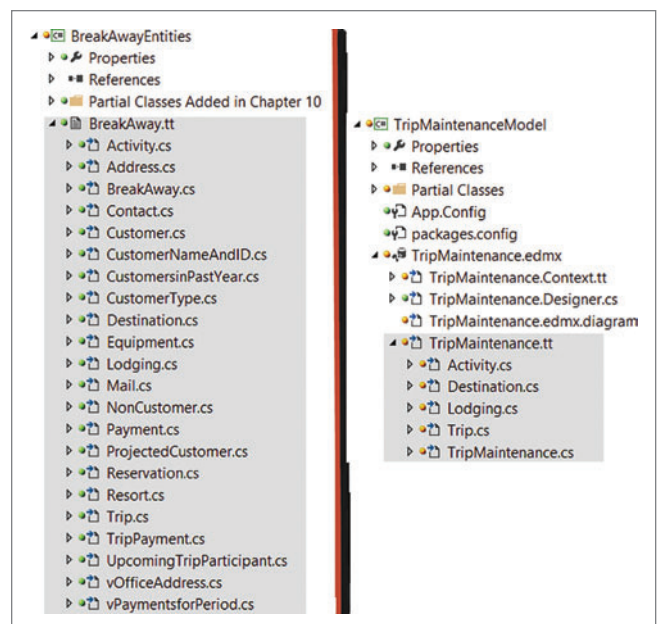


Figure 3 The Original Classes Generated from the T4 Template for the Large Model, Compared to the New Model and Its Generated Classes

We didn't invent the Internet...

...but our components help you power the apps that bring it to business.



TOOLS • COMPONENTS • ENTERPRISE ADAPTERS

- **E-Business**
AS2, EDI/X12, NAESB, OFTP ...
- **Credit Card Processing**
Authorize.Net, TSYS, FDMS ...
- **Shipping & Tracking**
FedEx, UPS, USPS ...
- **Accounting & Banking**
QuickBooks, OFX ...
- **Internet Business**
Amazon, eBay, PayPal ...
- **Internet Protocols**
FTP, SMTP, IMAP, POP, WebDav ...
- **Secure Connectivity**
SSH, SFTP, SSL, Certificates ...
- **Secure Email**
S/MIME, OpenPGP ...
- **Network Management**
SNMP, MIB, LDAP, Monitoring ...
- **Compression & Encryption**
Zip, Gzip, Jar, AES ...



The Market Leader in Internet Communications, Security, & E-Business Components

Each day, as you click around the Web or use any connected application, chances are that directly or indirectly some bits are flowing through applications that use our components, on a server, on a device, or right on your desktop. It's your code and our code working together to move data, information, and business. We give you the most robust suite of components for adding Internet Communications, Security, and E-Business Connectivity to

any application, on any platform, anywhere, and you do the rest. Since 1994, we have had one goal: to provide the very best connectivity solutions for our professional developer customers. With more than 100,000 developers worldwide using our software and millions of installations in almost every Fortune 500 and Global 2000 company, our business is to connect business, one application at a time.

connectivity
powered by 

To learn more please visit our website →

www.nsoftware.com

Fixing Code in the Context's Custom Partial Classes I began my fix-up by building the project that contains my new model. Once this project is sorted out, I can attack projects that depend on it. The partial class I originally created to extend the ObjectContext was the first to fail.

One of the custom methods in the partial class is `ManagedEntities`, which helped me see what entities were being tracked by the context. `ManagedEntities` relied on an extension method I created: a parameterless overload of `GetObjectStateEntries`. My use of that overload was the cause of the compiler error:

```
public IEnumerable<T> ManagedEntities<T>() {  
    var oses = ObjectStateManager.GetObjectStateEntries();  
    return oses.Where(entry => entry.Entity is T)  
                .Select(entry => (T)entry.Entity);  
}
```

Rather than fixing the underlying `GetObjectStateEntries` extension method, I can just eliminate it and the `ManagedEntities` method because the `DbContext` API has a method on the `DbSet` class called `Local` I can use instead.

There are two approaches I could take to this refactor. One is to find all of the code using my new model that calls `ManagedEntities` and replace it with the `DbSet.Local` method. Here's an example of code that uses `ManagedEntities` to iterate through all of the `Trips` the context is tracking:

```
foreach (var trip in _context.ManagedEntities<Trip>())
```

I could replace that with:

```
foreach (var trip in _context.Trips.Local.ToList())
```

Note that `Local` returns an `ObservableCollection`, so I add `ToList` to extract the entities from that.

I'm ready to attack the
harder chore: replacing
my ObjectContext with the
DbContext.

Alternatively, if my code has many calls to `ManagedEntities`, I could just change the logic behind `ManagedEntities` and avoid having to edit all the places it's used. Because the method is generic, it's not as straightforward as simply using the `Trips DbSet`, but the change is still simple enough:

```
public IEnumerable<T> ManagedEntities<T>() where T : class {  
    return Set<T>().Local.ToList();  
}
```

Most important, my `Trip Management` logic is no longer depending on the overloaded `GetObjectStateEntries` extension method. I can leave that method intact so that my original `ObjectContext` can continue to use it.

In the long run, many of the tricks I had to perform with extension methods became unnecessary when using the `DbContext` API. They were such commonly needed patterns that the `DbContext` API includes simple ways, like the `Local` method, to perform them.

The next broken method I found in the partial class was one I used to set an entity's tracked state to `Modified`. Again, I had been

Figure 4 My `IsTracked` Helper Method Became Unnecessary Thanks to the New `DbSet.Local` Method

```
public static bool IsTracked<TEntity>(this ObjectContext context,  
    Expression<Func<TEntity, object>> keyProperty, int keyId) where TEntity : class  
{  
    var keyPropertyName =  
        ((keyProperty.Body as UnaryExpression).Operand as MemberExpression).Member.Name;  
    var set = context.CreateObjectSet<TEntity>();  
    var entitySetName = set.EntitySet.EntityContainer.Name + "." + set.EntitySet.Name;  
    var key = new EntityKey(entitySetName, keyPropertyName, keyId);  
    ObjectStateEntry ose;  
    if (context.ObjectStateManager.TryGetObjectStateEntry(key, out ose))  
    {  
        return true;  
    }  
    return false;  
}
```

forced to use some non-obvious EF code to do the deed. The line of code that fails is:

```
ObjectStateManager.ChangeObjectState(entity, EntityState.Modified);
```

I can replace this with the more straightforward `DbContext.Entry().State` property. Because this code is in the partial class that extends my `DbContext`, I can access the `Entry` method directly rather than from an instance of `TripMaintenanceContext`. The new line of code is:

```
Entry(entity).State = EntityState.Modified;
```

The final method in my partial class—also broken—uses the `ObjectSet.ApplyCurrentValues` method to fix the state of a tracked entity using the values of another (untracked) instance of the same type. `ApplyCurrentValues` uses the identity value of the instance you pass in, finds the matching entity in the change tracker and then updates it using the values of the passed in object.

There's no equivalent in `DbContext` to `ApplyCurrentValues`. `DbContext` allows you to do a similar value replacement using `Entry().CurrentValues().Set`, but this requires that you already have access to the tracked entity. There's no easy way to build a generic method to find that tracked entity to replace that functionality. However, all is not lost. You can just keep using the special `ApplyCurrentValues` method by leveraging the ability to access `ObjectContext` logic from a `DbContext`. Remember, `DbContext` is a wrapper around `ObjectContext` and the API provides a way to drill down to the underlying `ObjectContext` for special cases, such as the `IObjectContextAdapter`. I added a simple property, `Core`, to my partial class to make it easy to reuse:

```
public ObjectContext Core {  
    get {  
        return (this as IObjectContextAdapter).ObjectContext;  
    }  
}
```

I then modified the relevant method of my partial class to continue to use `ApplyCurrentValues`, calling `CreateObjectSet` from the `Core` property. This gives me an `ObjectSet` so I can continue to use `ApplyCurrentValues`:

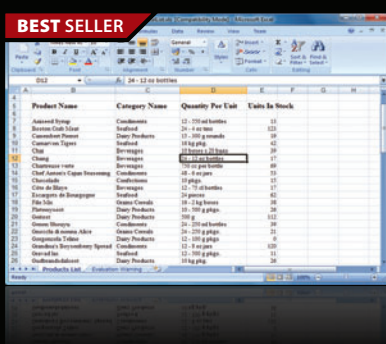
```
public void UpdateEntity<T>(T modifiedEntity) where T : class {  
    var set = Core.CreateObjectSet<T>();  
    set.ApplyCurrentValues(modifiedEntity);  
}
```

With this last change, my model project was able to compile. Now it was time to deal with broken code in the layers between my UI and model.

**Help & Manual Professional** | from \$583.10

Easily create documentation for Windows, the Web and iPad.

- Powerful features in an easy accessible and intuitive user interface
- As easy to use as a word processor, but with all the power of a true WYSIWYG XML editor
- Single source, multi-channel publishing with conditional and customized output features
- Output to HTML, WebHelp, CHM, PDF, ePub, RTF, e-book or print
- Styles and Templates give you full design control

**Aspose.Total for .NET** | from \$2,449.02

Every Aspose .NET component in one package.

- Programmatically manage popular file formats including Word, Excel, PowerPoint and PDF
- Work with charts, diagrams, images, project plans, emails, barcodes, OCR and OneNote files alongside many more document management features in .NET applications
- Common uses also include mail merging, adding barcodes to documents, building dynamic reports on the fly and extracting text from PDF files

**LEADTOOLS Document Imaging SDKs V18** | from \$2,695.50

Add powerful document imaging functionality to desktop, tablet, mobile & web applications.

- Comprehensive document image cleanup, preprocessing, viewer controls and annotations
- Fast and accurate OCR, OMR, ICR and Forms Recognition with multi-threading support
- PDF & PDF/A Read / Write / Extract / View / Edit
- Barcode Detect / Read / Write for UPC, EAN, Code 128, Data Matrix, QR Code, PDF417
- Zero-Footprint HTML5/JavaScript Controls & Native WinRT Libraries for Windows Store

**ComponentOne ActiveReports 8** | from \$1,567.02

The award-winning .NET reporting tool for HTML5, WPF, WinForms, ASP.NET & Windows Azure.

- Create sophisticated, fast and powerful reports
- Generate flexible layouts using Section, Region and Fixed page designers
- Experience a new scalable, distributed and load balanced Enterprise-grade report server
- Utilize the just released HTML5 and touch-optimized report viewers for mobile devices
- Explore an array of data visualization options including charts, maps and more

MergeOption.NoTracking to AsNoTracking Queries, and Lambdas, Too Specifying that EF should not track results is an important way to avoid unnecessary processing and improve performance. There are multiple places in my app where I query for data that will be used only as a reference list. Here's an example where I retrieve a list of read-only Trips along with their Destination information to be displayed in the ListBox on my UI. With the old API, you had to set the MergeOption on a query before executing it. Here's the ugly code I had to write:

```
var query = _context.Trips.Include("Destination");
query.MergeOption = MergeOption.NoTracking;
_trips = query.OrderBy(t => t.Destination.Name).ToList();
```

DbSet has a simpler way to do this using its AsNoTracking method. While I'm at it, I can get rid of the string in the Include method, as DbContext finally added the ability to use a lambda expression there. Here's the revised code:

```
_trips = _context.Trips.AsNoTracking().Include(t => t.Destination)
    .OrderBy(t => t.Destination.Name)
    .ToList();
```

DbSet.Local to the Rescue Again A number of places in my code required that I find out if an entity was being tracked when all I had was its identity value. I wrote a helper method, shown in **Figure 4**, to do this, and you can see why I wanted to encapsulate this code. Don't bother trying to decipher it; it's headed for the trash bucket.

Here's an example of code in my app that called IsTracked passing the value of a Trip I wanted to find:

```
_context.IsTracked<Trip>(t => t.TripID, tripID)
```

Thanks to the same DbSet.Local method I used earlier, I was able to replace that with:

```
_context.Trips.Local.Any(d => d.TripID == tripID))
```

And then I was able to delete the IsTracked method! Are you keeping track of how much code I've been able to delete so far?

Another method I had written for the application, AddActivity, needed to do more than just verify whether an entity was already being tracked. It needed to get that entity—another task Local can help me with. The AddActivity method (**Figure 5**) adds an Activity to a particular trip using ugly and non-obvious code that I had to write for theObjectContext API. This involves a many-to-many relationship between Trip and Activity. Attaching an activity instance to a trip that's being tracked causes EF to start tracking the activity, so I needed to protect the context from a duplicate—and my app from the resulting exception. In my method, I attempted to retrieve the entity's ObjectStateEntry. The TryGetObjectStateEntry performs two tricks at once. First, it returns a Boolean if the entry is found

and, second, returns either the found entry or null. If the entry wasn't null, I used its entity to attach to the trip; otherwise, I attached the one passed into my AddActivity method. Just describing that is exhausting.

Specifying that EF should not track results is an important way to avoid unnecessary processing and improve performance.

I thought long and hard about an efficient way to perform this logic but ended up with code of about the same length. Remember, this is a many-to-many relationship and they do require more care. However, the code is easier to write and easier to read. You don't have to tangle with the ObjectStateManager at all. Here's the part of the method I updated to use a similar pattern as I did earlier with the Destination:

```
var trackedActivity = _context.Activities.Local
    .FirstOrDefault(a => a.ActivityID ==
        activity.ActivityID);
if (trackedActivity != null) {
    activity = trackedActivity;
}
else {
    _context.Activities.Attach(activity);
}
```

Mission Complete

With this last fix, all of my tests passed and I was able to successfully use all of the features of the Trip Maintenance form. Now it's time to look for opportunities to take advantage of new EF6 features.

More important, any further maintenance on this portion of my application will be simplified because many tasks that were difficult with theObjectContext are so much easier with the DbContext API. Focusing on this small model, I've learned a lot that I can leverage for any otherObjectContext-to-DbContext transformations and I'm much better prepared for the future.

Equally important is to be smart about choosing what code should be updated and what should be left alone. I usually target features I know I'll have to maintain in the future and don't want to have to worry about interacting with the more difficult API. If you have code that will never be touched again and continues to work as EF evolves, I'd certainly think twice before diving into this challenge. Even if it's for the best, a broken arm can be quite painful. ■

Figure 5 The Original AddActivity Method Using the ObjectContext API

```
public void AddActivity(Activity activity)
{
    if (_context.GetEntityState(activity) == EntityState.Detached) {
        ObjectStateEntry existing0se;
        if (_context.ObjectStateManager
            .TryGetObjectStateEntry(activity, out existing0se))
        {
            activity = existing0se.Entity as Activity;
        }
        else {
            _context.Activities.Attach(activity);
        }
    }
    _currentTrip.Activities.Add(activity);
}
```

JULIE LERMAN is a Microsoft MVP, .NET mentor and consultant who lives in the hills of Vermont. You can find her presenting on data access and other.NET topics at user groups and conferences around the world. She blogs at thedatafarm.com/blog and is the author of "Programming Entity Framework" (2010) as well as a Code First edition (2011) and a DbContext edition (2012), all from O'Reilly Media. Follow her on Twitter at twitter.com/julielerman and see her Pluralsight courses at julieme/PS-Videos.

THANKS to the following Microsoft technical expert for reviewing this article: Andrew Oakley



**5 GREAT
CONFERENCES.**
1 GREAT PRICE.

**SAVE
\$500**

Register with code
L360JUL1 by August 6.



Scan the QR code
to register or
for more event
details.

live360events.com

IT AND DEVELOPER TRAINING THAT'S OUT OF THIS WORLD.

Live! 360 brings together five conferences, and the brightest minds in IT and Dev, to explore leading edge technologies and conquer current ones. These co-located events will incorporate knowledge transfer and networking, along with out-of-this-world education and training, as you create your own custom conference, mixing and matching sessions and workshops to best suit your needs.

Visual Studio **LIVE!**
EXPERT SOLUTIONS FOR .NET DEVELOPERS

ORLANDO
ROYAL PACIFIC RESORT AT
UNIVERSAL ORLANDO

**NOV
17-21**



"HELP US VISUAL STUDIO LIVE! – YOU'RE OUR ONLY HOPE!"

THE CODE IS STRONG WITH THIS ONE!

While your development challenges may not be as dire as the Rebellion's, we know you're searching for real-world knowledge and education on the Microsoft Platform. And for 21 years and counting, Visual Studio Live! has been the conference more developers trust for independent, practical training.

EVENT PARTERS

Magenic

Microsoft

PLATINUM SPONSOR



GOLD SPONSOR



SUPPORTED BY



Visual Studio
MAGAZINE



5 GREAT
CONFERENCES.
1 GREAT PRICE.



TECH EVENTS WITH PERSPECTIVE

Visual Studio Live! Orlando is part of Live! 360, the ultimate IT and Developer line-up. This means you'll have access to four (4) other co-located events at no additional cost:

SQL Server **LIVE!**
TRAINING FOR DBAs AND IT PROS

SharePoint **LIVE!**
TRAINING FOR COLLABORATION

ModernApps **LIVE!**
MOBILE, CROSS-DEVICE & CLOUD DEVELOPMENT

TECHMENTOR
IN-DEPTH TRAINING FOR IT PROS

WHETHER YOU ARE AN:

- Engineer
- Developer
- Programmer
- Software Architect
- Software Designer




You will walk away from this event having expanded your .NET skills and the ability to build better applications.

**REGISTER TODAY
AND SAVE \$500!**

Use promo code ORLJUL2 by August 6



CONNECT WITH VISUAL STUDIO LIVE!

-  [twitter.com/@vslive](https://twitter.com/vslive)
-  facebook.com – Search: "VS Live"
-  linkedin.com – Join the "Visual Studio Live" group!

Five (5) events, 29 tracks, and hundreds of sessions to choose from – mix and match sessions to create your own, custom event line-up – it's like no other conference available today!

Scaling Your Web Application with Azure Web Sites

Yochay Kiriaty

Surprisingly, scale is an often overlooked aspect of Web application development. Typically, the scale of a Web application becomes a concern only when things start to fail and the UX is compromised due to slowness or timeouts at the presentation layer. When a Web application starts exhibiting such performance deficits, it has reached its scalability point—the point at which lack of resources, such as CPU, memory or bandwidth, rather than a logical bug in the code, hampers its ability to function.

This is the time to scale your Web application and give it extra resources, whether more compute, additional storage or a stronger database back end. The most common form of scaling in the cloud is horizontal—adding additional compute instances that allow a Web application to run simultaneously on multiple Web servers

(instances). Cloud platforms, such as Microsoft Azure, make it very easy to scale the underlying infrastructure that supports your Web application by supplying any number of Web servers, in the form of virtual machines (VMs), at the flick of your finger. However, if your Web application isn't designed to scale and run across multiple instances, it won't be able to take advantage of the extra resources and will not yield the expected results.

This article takes a look at key design concepts and patterns for scaling Web applications. The implementation details and examples focus on Web applications running on Microsoft Azure Web Sites.

Before I start, it's important to note that scaling a Web application is very much dependent on the context and on the way your application is architected. The Web application used in this article is simple, yet it touches the fundamentals of scaling a Web application, specifically addressing scale when running on Azure Web Sites.

There are different levels of scale that cater to different business needs. In this article, I'll look at four different levels of scaling capabilities, from a Web application that can't run on multiple instances, to one that can scale across multiple instances—even across multiple geographical regions and datacenters.

Step One: Meet the Application

I'm going to start by reviewing the limitations of the sample Web application. This step will set the baseline from which the required modifications to enhance the scalability of the application will be made. I chose to modify an existing application, because in

This article discusses:

- The limitations of the sample Photo Gallery Web application
- Making the Photo Gallery into a stateless application
- Improving the way the photos are stored and served
- Enabling the application to run across multiple datacenters

Technologies discussed:

WebMatrix Photo Gallery Template for ASP.NET Web Pages, Azure SQL Database, Azure Web Sites, Azure Storage, Azure CDN

Code download available at:

msdn.microsoft.com/magazine/msdnmag0714

real life that's often what you're asked to do, rather than creating a brand-new application and design from scratch.

The application I'll use for this article is the WebMatrix Photo Gallery Template for ASP.NET Web Pages (bit.ly/1lIAJdQ). This template is a great way to learn how to use ASP.NET Web Pages to create real-world Web applications. It is a fully functional Web application that enables users to create photo albums and upload images. Anyone can view images, and logged-in users can leave comments. The Photo Gallery Web application can be deployed to Azure Web Sites from WebMatrix, or directly from the Azure Portal via the Azure Web Sites Gallery.

Looking closely at the Web application code reveals at least three significant architecture issues that limit the application's scalability: the use of a local SQL Server Express as the database; the use of an In-Process (local Web server memory) session state; and the use of the local file system to store photos.

I'll review each of these limitations in-depth.

The PhotoGallery.sdf file, found in the App_Data folder, is the default SQL Server Express database that's distributed with the application. SQL Server Express makes it easy to start developing an application and offers a great learning experience, but it also imposes serious restrictions on the application's ability to scale. A SQL Server Express database is, essentially, a file in the file system. The Photo Gallery application in its current state can't safely scale across multiple instances. Trying to scale across multiple instances can leave you with multiple instances of the SQL Server Express database file, each one a local file and likely out of sync with the others. Even if all Web server instances share the same file system, the SQL Server Express file can get locked by any one instance at different times, causing the other instances to fail.

The Photo Gallery application is also limited by the way it manages a user's session state. A session is defined as a series of requests issued by the same user within a certain period of time, and is managed by associating a session ID with each unique user. The ID is used for each subsequent HTTP request and is provided by the client, either in a cookie or as a special fragment of the request URL. The session data is stored on the server side in one of the supported session state stores, which include in-process memory, a SQL Server database or ASP.NET State Server.

Figure 1 **GetCurrentUser**

```
public static string GetCurrentUser(this HttpRequestBase request,
                                   HttpResponseBase response = null)
{
    string userName;
    try
    {
        if (request.Cookies["GuidUser"] != null)
        {
            userName = request.Cookies["GuidUser"].Value;
            var db = Database.Open("PhotoGallery");
            var guidUser = db.QuerySingle(
                "SELECT * FROM GuidUsers WHERE UserName = @0", userName);
            if (guidUser == null || guidUser.TotalLikes > 5)
            {
                userName = CreateNewUser();
            }
        }
        ...
        ...
    }
}
```

The Photo Gallery application uses the WebMatrix WebSecurity class to manage a user's login and state, and WebSecurity uses the default ASP.NET membership provider session state. By default, the session state mode of the ASP.NET membership provider is in-process (InProc). In this mode, session state values and variables are stored in memory on a local Web server instance (VM). Having user session state stored locally per Web server limits the ability of the application to run on multiple instances because subsequent HTTP requests from a single user can end up on different instances of Web servers. Because each Web server instance keeps its own copy of the state in its own local memory, you can end up with different InProc session state objects on different instances for the same users. This can lead to unexpected and inconsistent UXes. Here, you can see the WebSecurity class being used to manage a user's state:

```
_AppStart.cshtml
@{
    WebSecurity.InitializeDatabaseConnection
        ("PhotoGallery", "UserProfiles", "UserId", "Email", true);
}

Upload.cshtml
@{
    WebSecurity.RequireAuthenticatedUser();
    ...
}
```

The WebSecurity class is a helper, a component that simplifies programming in ASP.NET Web Pages. Behind the scenes, the WebSecurity class interacts with an ASP.NET membership provider, which in turn performs the lower-level work required to perform security tasks. The default membership provider in ASP.NET Web Pages is the SimpleMembershipProvider class and, by default, its session state mode is InProc.

Finally, the current version of the Photo Gallery Web application stores photos in the database, each photo as an array of bytes. Essentially, because the application is using SQL Server Express, photos are saved on the local disk. For a photo gallery application, one of the main scenarios is viewing photos, so the application may need to handle and show a great many photo requests. Reading photos from a database is less than ideal. Even using a more sophisticated database, such as SQL Server or Azure SQL Database, isn't ideal, mainly because retrieving photos is such an expensive operation.

In short, this version of Photo Gallery is a stateful application, and stateful applications do not scale well across multiple instances.

Step Two: Modifying Photo Gallery To Be a Stateless Web Application

Now that I've explained some of the limitations of the Photo Gallery application with regard to scaling, I'll address them one by one to improve the application's scale capabilities. In Step Two, I'll make the necessary changes to convert the Photo Gallery from stateful to stateless. At the end of Step Two, the updated Photo Gallery application will be able to safely scale and run across multiple Web server instances (VMs).

First, I'll replace SQL Server Express with a more powerful database server—Azure SQL Database, a cloud-based service from Microsoft that offers data-storage capabilities as part of the Azure services platform. Azure SQL Database Standard and Premium

SKUs offer advanced business continuity features I'll use in Step Four. For now, I'll just migrate the database from SQL Server Express to Azure SQL Database. You can do this easily using the WebMatrix database migration tool, or any other tool you want, to convert the SDF file into the Azure SQL Database format.

As long as I'm already migrating the database, it's a good opportunity to make some schema modifications that, though small, will have significant impact on the application's scaling capabilities.

First, I'll convert the ID column type of some of the tables (Galleries, Photos, UserProfiles and so on) from INT to GUID. This change will prove useful in Step Four, when I update the application to run across multiple regions and need to keep the database and photo content in sync. It's important to note that this modification doesn't force any code changes on the application; all SQL queries in the application remain the same.

Next, I'll stop storing photos as arrays of bytes in the database. This change involves both schema and code modifications. I'll remove the FileContents and FileSize columns from the Photos table, store photos directly to disk, and use the photo ID, which is now a GUID, as means of distinguishing photos.

The following code snippet shows the INSERT statement before the change (note that both fileBytes and fileBytes.Length are stored directly in the database):

```
db.Execute(@"INSERT INTO Photos
(Id, GalleryId, UserName, Description, FileName, FileExtension,
ContentType, FileSize, UploadDate, FileContents, Likes)
VALUES (@0, @1, @2, @3, @4, @5, @6, @7, @8, @9, @10)",
guid.ToString(), galleryId, Request.GetCurrentUser(Response), "",
fileName, fileExtension, fileUpload.ImageFormat, fileBytes.Length,
DateTime.Now, fileBytes, 0);
```

And here's the code after the database changes:

```
using (var db = Database.Open("PhotoGallery"))
{
    db.Execute(@"INSERT INTO Photos
(Id, GalleryId, UserName, Description, FileName, FileExtension,
UploadDate, Likes)
VALUES (@0, @1, @2, @3, @4, @5, @6, @7)", imageId, galleryId,
userName, "", imageId, extension,
DateTime.UtcNow, 0);
}
```

In Step Three, I'll explore in more detail how I modified the application. For now, suffice it to say the photos are saved to a central location, such as a shared disk, that all Web server instances can access.

The last change I'll make in Step Two is to stop using the InProc session state. As noted earlier, WebSecurity is a helper class that interacts with ASP.NET membership providers. By default, ASP.NET Simple-Membership session state mode is InProc. There are several out-of-process options you can use with SimpleMembership, including SQL Server and ASP.NET State Server service. These two options enable session state to be shared among multiple Web server

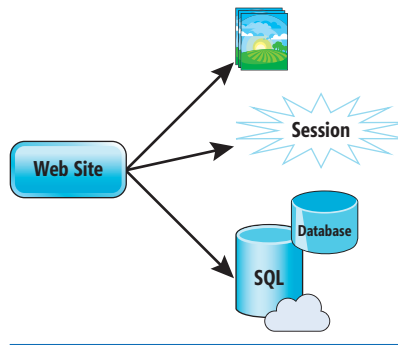


Figure 2 Logical Representation of the Modified Photo Gallery Application

instances and avoid server affinity; that is, they don't require the session to be tied to one specific Web server.

My approach also manages state out of process, specifically using a database and cookie. However, I rely on my own implementation rather than ASP.NET, essentially because I want to keep things simple. The implementation uses a cookie and stores the session ID and its state in the database. Once a user logs in, I assign a new GUID as a session ID, which I store in the database. That GUID is also returned to the user in the form of a cookie. The following code shows the

CreateNewUser method, which is called every time a user logs in:

```
private static string CreateNewUser()
{
    var newUser = Guid.NewGuid();
    var db = Database.Open("PhotoGallery");
    db.Execute(@"INSERT INTO GuidUsers (UserName, TotalLikes) VALUES (@0, @1)",
    newUser.ToString(), 0);
    return newUser.ToString();
}
```

When responding to an HTTP request, the GUID is embedded in the HTTP response as a cookie. The username passed to the AddUser method is the product of the CreateNewUser function just shown, like so:

```
public static class ResponseExtensions
{
    public static void AddUser(this HttpResponseBase response, string userName)
    {
        var userCookie = new HttpCookie("GuidUser")
        {
            Value = userName,
            Expires = DateTime.UtcNow.AddYears(1)
        };
        response.Cookies.Add(userCookie);
    }
}
```

When handling an incoming HTTP request, first I try to extract the user ID, represented as a GUID, from the GuidUser cookie. Next, I look for that userID (GUID) in the database and extract any user-specific information. **Figure 1** shows part of the GetCurrentUser implementation.

Both CreateNewUser and GetCurrentUser are part of the Request-Extensions class. Similarly, AddUser is part of the ResponseExtensions class. Both classes plug into the ASP.NET request-processing pipeline, handling requests and responses, respectively.

My approach to managing session state is a rather a naïve one as it isn't secure and doesn't enforce any authentication. However, it

shows the benefit of managing sessions out of process, and it scales. When you implement your own session-state management, whether or not you base it on ASP.NET, make sure you use a secure solution that includes authentication and a secure way to encrypt the cookie you return.

At this point, I can safely claim the updated Photo Gallery application is now a stateless Web application. By replacing the local SQL Server Express database implementation with

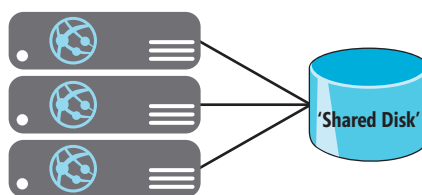


Figure 3 With Microsoft Azure Web Sites, All Instances of a Web Application See the Same Shared Disk

Spreadsheets Made Easy.



Fastest Calculations

Evaluate complex Excel-based models and business rules with the fastest and most complete Excel-compatible calculation engine available.



Comprehensive Charting

Enable users to visualize data with comprehensive Excel-compatible charting which makes creating, modifying, rendering and interacting with complex charts easier than ever before.



Windows
Forms



Silverlight



WPF

Powerful Controls

Add powerful Excel-compatible viewing, editing, formatting, calculating, filtering, sorting, charting, printing and more to your WinForms, WPF and Silverlight applications.



Scalable Reporting

Easily create richly formatted Excel reports without Excel from any ASP.NET, Windows Forms, WPF or Silverlight application.

Download your free fully functional evaluation at SpreadsheetGear.com



SpreadsheetGear

Toll Free USA (888) 774-3273 | Phone (913) 390-4797 | sales@spreadsheetgear.com

Azure SQL Database, and changing the session state implementation from InProc to out of process, using a cookie and a database, I successfully converted the application from stateful to stateless, as illustrated in **Figure 2**.

Taking the necessary steps to ensure your Web application is stateless is probably the most meaningful task during the development of a Web application. The ability to safely run across multiple Web server instances, with no concerns about user state, data corruption or functional correctness, is one of the most important factors in scaling a Web application.

Step Three: Additional Improvements That Go a Long Way

Changes made to the Photo Gallery Web application in Step Two ensure the application is stateless and can safely scale across multiple Web server instances. Now, I'll make some additional improvements that can further enhance the application's scalability characteristics, enabling the Web application to handle larger loads with fewer resources. In this step, I'll review storage strategies, and address async design patterns that enhance UX performance.

One of the changes discussed in Step Two was saving photos to a central location, such as a shared disk that all Web server instances could access, rather than to a database. Azure Web Sites architecture ensures that all instances of a Web application running across multiple Web servers share the same disk, as **Figure 3** illustrates.

From the Photo Gallery Web application perspective, "shared disk" means when a photo gets uploaded by a user, it's saved to the .../uploaded folder, which looks like a local folder. However, when the image is written to disk, it isn't saved "locally" on the specific Web server that handles the HTTP request, but instead saved to a central location that all Web servers can access. Therefore, any server can write any photo to the shared disk and all other Web servers can read that image. The photo metadata is stored in the database and used by the application to read the photo ID—a GUID—and return the image URL as part of the HTML response. The following code snippet is part of view.cshtml, which is the page I use to enable viewing images:

```

```

The source of the image HTML element is populated by the return value of the GetFullImageUrl helper function, which takes a photo ID and file extension (.jpg, .png and so on) and returns a string representing the URL of the image.

Saving the photos to a central location ensures that the Web application is stateless. However, with the current implementation, a given image is served directly from one of the Web servers running the Web application. Specifically, the URL of each image source points to the Web application's URL. As a result, the image itself is served from one of the Web servers running the Web application, meaning the actual bytes of the image are sent as an HTTP response from the Web

server. This means your Web server, on top of handling dynamic Web pages, also serves static content, such as images. Web servers can serve a lot of static content at great scale, but doing so imposes a toll on resources, including CPU, IO and memory. If you could make sure only static content, such as photos, isn't served directly from the Web server running your Web application, but rather from somewhere else, you could reduce the number of HTTP requests hitting the Web servers. By doing so, you'd free resources on the Web server to handle more dynamic HTTP requests.

The first change to make is to use Azure Blob storage (bit.ly/TOK3yb) to store and serve user photos. When a user asks to view an image, the URL returned by the updated GetFullImageUrl points to an Azure Blob. The end result looks like the following HTML, where the image URL points to Blob storage:

```

```

This means an image is served directly from the Blob storage and not from the Web servers running the Web application.

In contrast, the following shows photos saved to an Azure Web Sites shared disk:

```

```

The Photo Gallery Web application uses two containers, full and thumbnail. As you'd expect, full stores photos in their original sizes, while thumbnail stores the smaller images that are shown in the gallery view.

```
public static string GetFullImageUrl(string imageId, string imageExtension)
{
    return String.Format("{0}/full/{1}{2}",
        Environment.ExpandEnvironmentVariables("%AZURE_STORAGE_BASE_URL%"),
        imageId, imageExtension);
}
```

AZURE_STORAGE_BASE_URL is an environment variable that contains the base URL for the Azure Blob, in this case, - http://photogalcontentwestus.blob.core.windows.net. This environment variable can be set in the Azure Portal at the Site Config tab, or it can be part of the application web.config. Setting environment variables from the Azure Portal gives you more flexibility, however, because it's easier to change without the need to redeploy.

Azure Storage is used in much the same way as a content delivery network (CDN), mainly because HTTP requests for images aren't being served from the application's Web servers, but directly from an Azure Storage container. This substantially reduces the amount of

static HTTP request traffic that ever reaches your Web servers, allowing the Web servers to handle more dynamic requests. Note also that Azure Storage can handle far more traffic than your average Web server—a single container can scale to serve many tens of thousands of requests per second.

In addition to using Blob storage for static content, you can also add the Microsoft Azure CDN. Adding a CDN on top of your Web application further improves performance, as the CDN will serve all static content. Requests for a photo already cached on the CDN won't reach the Blob storage.

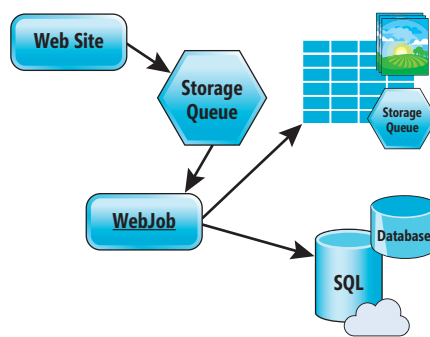


Figure 4 Logical Representation of Photo Gallery Post Step Three

Creating a report is as easy as writing a letter



Reuse MS Word documents as your reporting templates



Create encrypted and print-ready Adobe PDF and PDF/A



Royalty-free WYSIWYG template designer



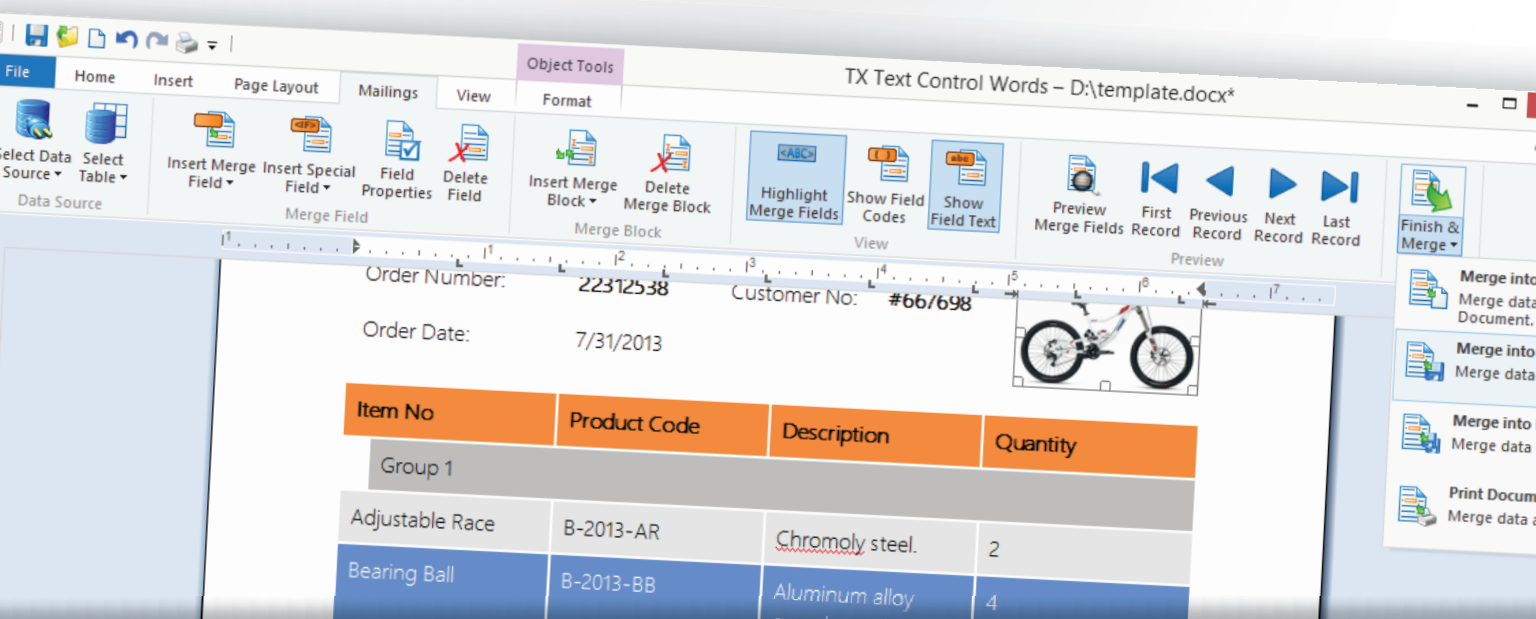
Powerful and dynamic 2D/3D charting support



Easy database connections and master-detail nested blocks



1D/2D barcode support including QRCode, IntelligentMail, EAN



www.textcontrol.com/reporting



txtextcontrol

US: +1 855-533-TEXT
EU: +49 421 427 067-10



Visual Studio

Microsoft

Partner

Reporting

Rich Text Editing

Spell Checking

Barcodes

PDF Reflow

Figure 5 Reading a Message from a Queue and Updating the Database

```
public static void ProcessUploadQueueMessages
([QueueInput("uploadqueue")] string queueMessage, IBinder binder)
{
    var splitted = queueMessage.Split(',').Select(m => m.Trim()).ToArray();
    var userName = splitted[0];
    var galleryId = splitted[1];
    var imageId = splitted[2];
    var extension = splitted[3];
    var filePath = Path.Combine(ImageFolderPath, imageId + extension);

    UploadFullImage(filePath, imageId + extension, binder);
    UploadThumbnail(filePath, imageId + extension, binder);
    SafeGuard(() => File.Delete(filePath));

    using (var db = Database.Open("PhotoGallery"))
    {
        db.Execute(@"INSERT INTO Photos Id, GalleryId, UserName,
        Description, FileName, FileExtension, UploadDate, Likes)
        VALUES @0, @1, @2, @3, @4, @5, @6, @7)", imageId,
        galleryId, userName, "", imageId, extension, DateTime.UtcNow, 0);
    }
}
```

Moreover, a CDN also enhances perceived performance, as the CDN typically has an edge server closer to the end customer. The details of adding a CDN to the sample application are beyond the scope of this article, as the changes are mostly around DNS registration and configuration. But when you're addressing production at scale and you want to make sure your customers will enjoy a quick and responsive UI, you should consider using the CDN.

I haven't reviewed the code that handles a user's uploaded images, but this is a great opportunity to address a basic async pattern that improves both Web application performance and UX. This will also help data synchronization between two different regions, as you'll see in Step Four.

The next change I'll make to the Photo Gallery Web application is to add an Azure Storage Queue, as a way to separate the front end of the application (the Web site) from the back-end business logic (WebJob + database). Without a Queue, the Photo Gallery code handled both front end and back end, as the upload code saved the full-size image to storage, created a thumbnail and saved it to storage, and updated the SQL Server database. During that

time, the user waited for a response. With the introduction of an Azure Storage Queue, however, the front end just writes a message to the Queue and immediately returns a response to the user. A background process, WebJob (bit.ly/1mw0A3w), picks up the message from the Queue and performs the required back-end business logic. For Photo Gallery, this includes manipulating images, saving them to the correct location and updating the database. **Figure 4** illustrates the changes made in Step Three, including using Azure Storage and adding a Queue.

Now that I have a Queue, I need to change the upload.cshtml code. In the following code you can see that instead of performing complicated business logic with image manipulation, I use StorageHelper to enqueue a message (the message includes the photo ID, the photo file extension and gallery ID):

```
var file = Request.Files[i];
var fileExtension = Path.GetExtension(file.FileName).Trim();
guid = Guid.NewGuid();
using
var fileStream = new FileStream( Path.Combine( HostingEnvironment.
MapPath("~/App_Data/Upload/"), guid + fileExtension), FileMode.Create))
{
    file.InputStream.CopyTo(fileStream);
    StorageHelper.EnqueueUploadAsync(
        Request.GetCurrentUser(Response), galleryId, guid.ToString(), fileExtension);
}
```

StorageHelper.EnqueueUploadAsync simply creates a CloudQueueMessage and asynchronously uploads it to the Azure Storage Queue:

```
public static Task EnqueueUploadAsync
(string userName, string galleryId, string imageId, string imageExtension)
{
    return UploadQueue.AddMessageAsync(
        new CloudQueueMessage(String.Format("{0}, {1}, {2}, {3}",
        userName, galleryId, imageId,
        imageExtension)));
}
```

WebJob is now responsible for the back-end business logic. The new WebJobs feature of Azure Web Sites provides an easy way to run programs such as services or background tasks on a Web site. The WebJob listens for changes on the Queue and picks up any new message. The ProcessUploadQueueMessages method, shown in **Figure 5**, is called whenever there's at least one message in the Queue. The QueueInput attribute is part of the Microsoft Azure WebJobs SDK (bit.ly/1cN9eCx), which is a framework that simplifies the task of adding background processing to Azure Web Sites. The

WebJobs SDK is out of scope for this article, but all you really need to know is that the WebJob SDK lets you easily bind to a Queue, in my case uploadqueue, and listen to incoming messages.

Each message is decoded, splitting the input string into its individual parts. Next, the method calls two helper functions to manipulate and upload the images to the Blob containers. Last, the database is updated.

At this point, the updated Photo Gallery Web application can handle many millions of HTTP requests per day.

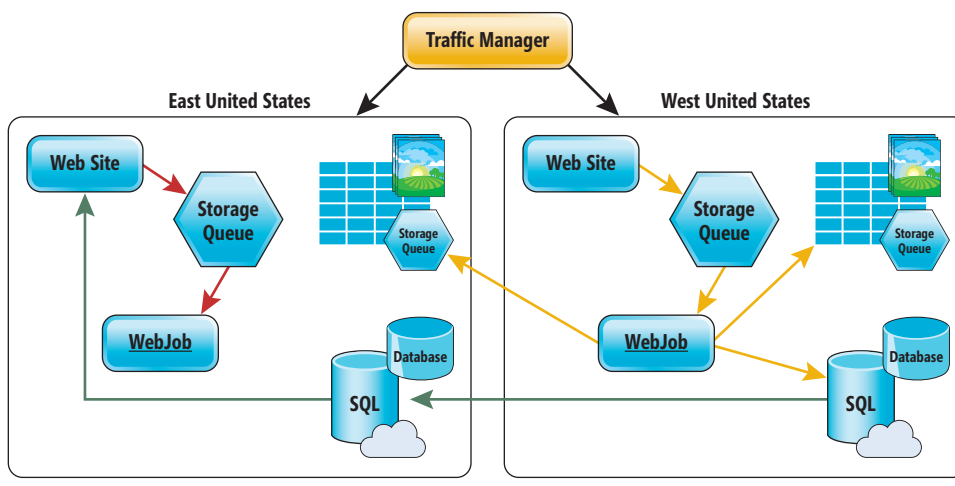


Figure 6 Logical Representation of Photo Gallery Post Step Four

Visual Studio **LIVE!**
EXPERT SOLUTIONS FOR .NET DEVELOPERS

REDMOND 2014

August 18 – 22 | Microsoft Headquarters | Redmond, WA



**SET YOUR COURSE
FOR 127.0.0.1!**



Topics include:

- Visual Studio/.NET Framework
- Windows Client
- JavaScript/HTML5 Client
- ASP.NET
- Cloud Computing
- Windows Phone
- Cross-Platform Mobile Development
- SharePoint
- SQL Server

**YOUR GUIDE TO THE
.NET DEVELOPMENT
UNIVERSE**

**SAVE
\$300**



Use promo code
REDJULTI by July 16

vslive.com/redmond

Visual Studio
EXPERT SOLUTIONS FOR .NET DEVELOPERS



REDMOND 2014

August 18 – 22 | Microsoft Headquarters | Redmond, WA



From August 18 – 22, 2014, developers, engineers, software architects and designers will land in Redmond, WA at the idyllic Microsoft headquarters for 5 days of cutting-edge education on the Microsoft Platform.

Come experience code at the source – rub elbows with Microsoft stars, get the inside scoop on what's next, and learn what you need to know now. Over 5 days and 60+ sessions and workshops, you'll explore the .NET Development Universe, receiving the kind of practical, unbiased training you can only get at Visual Studio Live!

vslive.com/redmond

SAVE \$300

vslive.com/redmond

CONNECT WITH VISUAL STUDIO LIVE!



twitter.com/vslive – @VSLive



facebook.com – Search "VSLive"



linkedin.com – Join the "Visual Studio Live" group!



Use promo code
REDJULTI by July 16

EVENT SPONSOR

Microsoft

PLATINUM SPONSOR



SPONSOR



SUPPORTED BY



PRODUCED BY



Step Four: Global Reach

I've already made tremendous improvements to the scaling ability of the Photo Gallery Web application. As I noted, the application can now handle many millions of HTTP requests using only a few large servers on Azure Web Sites. Currently, all these servers are located in a single Azure datacenter. While running from a single datacenter isn't exactly a scale limitation—at least not by the standard definition of scale—if your customers from around the globe require low latency, you'll need to run your Web application from more than one datacenter. This also improves your Web application's durability and business-continuity capabilities. In the rare case that one datacenter experiences an outage, your Web application will continue to serve traffic from the second location.

In this step, I'll make the changes to the application that enable it to run across multiple datacenters. For this article, I'll focus on running from two locations in an active-active mode, where the application in both datacenters allows the user to view photos, as well as upload photos and comments.

Keep in mind that because I understand the context for the Photo Gallery Web application, I know the majority of user operations are read operations, showing photos. Only a small number of user requests involve uploading new photos or updating comments. For the Photo Gallery Web application, I can safely state the read/write ratio is at least 95 percent reads. This allows me to make some assumptions, for example, that having eventual consistency across the system is acceptable, as is a slower response for write operations.

It's important to understand these assumptions are context-aware and depend on the specific characteristics of a given application, and will most likely change from one application to the other.

Surprisingly, the amount of work required to run Photo Gallery from two different locations is small, as most of the heavy lifting was done in Step Two and Step Three. **Figure 6** shows a high-level block diagram of the application topology running from two different datacenters. The application in West U.S. is the “main” application and basically has the output of Step Three. The application in East U.S. is the “secondary” site, and the Azure Traffic Manager is placed on top of both. Azure Traffic Manager has several configuration options. I'll use the Performance option, which causes Traffic Manager to monitor both sites for latency in their respective regions and route traffic based on the lowest latency. In this case, customers from New York (east coast) will be directed to the East U.S. site and customers from San Francisco (west coast) will be directed to the West U.S. site. Both sites are active at the same time, serving traffic. Should the application in one region experience performance issues, for whatever reason, the Traffic Manager will route traffic to the other application. Because the data is synced, no data should be lost.

I'll look at the changes to the West U.S. application. The only code change is to the WebJob listening for messages in the Queue. Instead of saving photos to one Blob, the WebJob saves photos to the local and “remote” Blob store. In **Figure 5**, UploadFullImage is a helper method that saves photos to Blob storage. To enable copying a photo to a remote Blob as well as a local Blob, I added the ReplicateBlob helper function at the end of UploadFullImage, as you can see here:

```
private static void UploadFullImage(
    string imagePath, string blobName, IBinder binder)
{
    using (var fileStream = new FileStream(imagePath, FileMode.Open))
    {
        using (var outputStream =
            binder.Bind<Stream>(new BlobOutputAttribute(
                String.Format("full/{0}", blobName))))
        {
            fileStream.CopyTo(outputStream);
        }
    }
    RemoteStorageManager.ReplicateBlob("full", blobName);
}
```

The ReplicateBlob method in the following code has one important line—the last line calls the StartCopyFromBlob method, which asks the service to copy all the contents, properties and metadata of a Blob to a new Blob (I let the Azure SDK and storage service take care of the rest):

```
public static void ReplicateBlob(string container, string blob)
{
    if (sourceBlobClient == null || targetBlobClient == null)
        return;
    var sourceContainer = sourceBlobClient.GetContainerReference(container);
    var targetContainer = targetBlobClient.GetContainerReference(container);
    if (targetContainer.CreateIfNotExists())
    {
        targetContainer.SetPermissions(sourceContainer.GetPermissions());
    }
    var targetBlob = targetContainer.GetBlockBlobReference(blob);
    targetBlob.StartCopyFromBlob(sourceContainer.GetBlockBlobReference(blob));
}
```

In East U.S., the ProcessLikeQueueMessages method doesn't process anything; it simply pushes the message to the West U.S. Queue. The message will be processed in the West U.S., images will be replicated as explained earlier and the database will get synced, as I'll explain now.

This is the last missing piece of magic—synchronizing the database. To achieve this I'll use the Active Geo-Replication (continuous copy) preview feature of Azure SQL Database. With this feature, you can have secondary read-only replicas of your master database. Data written to the master database is automatically copied to the secondary database. The master is configured as a read-write database and all secondary databases are read-only, which is the reason in my scenario that messages are pushed from the East U.S. Queue to West U.S. Once you configure Active Geo-Replication (via the Portal), the databases will be in sync. No code changes are required beyond what I've already covered.

Wrapping Up

Microsoft Azure lets you build Web apps that can scale a lot with very little effort. In this article, I showed how, in just a few steps, you can modify a Web application from one that can't really scale at all because it can't run on multiple instances, to one that can run not only across multiple instances, but also across multiple regions, handling millions (high tens of millions) of HTTP requests. The examples are specific to the particular application, but the concept is valid and can be implemented on any given Web application. ■

Yochay Kiriati is a principal program manager lead on the Microsoft Azure team, working on Azure Web Sites. Reach him at yochay@microsoft.com and follow him on Twitter at twitter.com/yochayk.

THANKS to the following Microsoft technical expert for reviewing this article:
Mohamed Ameen Ibrahim

Architect for the Cloud Using Microsoft Azure Web Sites

Apurva Joshi and Sunitha Muthukrishna

There's one important thing to keep in mind when designing a cloud solution—always design for failure. Many applications, however, aren't architected this way. The primary reason for this is a lack of awareness of how to design an architecture using Microsoft Azure Web Sites that's sufficiently resilient. So how do you build a cloud solution that's robust enough to handle failure? This article will discuss techniques you can employ to design such a system.

Single Web Site Instance

Azure Web Sites provides hosting plans on several tiers: Free, Shared, Basic and Standard. The Free and Shared tiers provide sites with a shared infrastructure, meaning your sites share resources with other sites. In the Basic and Standard tier, your sites are provided with a dedicated infrastructure, meaning that only the site or sites you choose to associate with your plan will run on those resources.

This article discusses:

- Creating highly available Web sites and applications
- Optimize databases, cache and the Web application layer for high availability
- Tune recovery options and deploy diagnostic tools to investigate and remediate issues

Technologies discussed:

Technologies discussed: Microsoft Azure, Microsoft .NET Framework, Azure SQL Database Premium, Kudu Console

At these tiers, you can configure your Web hosting plan to use one or more virtual machine (VM) instances. These tiers can support small, medium and large instances. The provider will manage these VMs on your behalf, meaning you'll never need to worry about OS updates or security and configuration updates.

For running a production-level Web site on Azure Web Sites, the Basic or Standard tier is recommended based on your application size and the amount of traffic to your Web site. Understanding your application requirements is a good starting point:

1. What components does your application need—database, e-mail provider, cache and so on?
2. What components are at risk of failure and need to be replicated?
3. What features do you need—SSL, staging slots and so on?
4. How much traffic should your Web site be able to manage?

With these answers, you can create a single Web site and add components such as database and cache to your application as needed. In **Figure 1**, you can see how you might architect a single Web site and its dependent components.

In the Single Web Site scenario, the biggest caveat is in the event of a service-related outage for any component (Web site, database or cache service), your Web site will be unavailable during the outage, so there will be an impact on your customers and your business.

This design doesn't consider the risks involved with cloud solutions, nor does it include a way to mitigate them. In a cloud environment, your design goal should be to create a highly available Web site that will minimize downtime and expedite recovery during an outage.

The Goal Is High Availability

A typical Web application stack in Azure Web Sites will consist of a Web application, database, Azure Storage and some form of cache. All of these components are tightly coupled to avoid a single entity or component becoming a single point of failure (SPOF). That's a key design criterion for architecting any cloud solution. The goal is:

- Avoid SPOF in your design
- Redundancy across each layer in your design

Azure Web Sites provides an SLA of 99.9 percent.

This means you can expect a downtime of about 10 minutes per week due to scheduled deployment or upgrades conducted by the Azure Web Sites service. Still, 10 minutes of downtime can have a great impact on your business. Your goal should be to design your solution to mitigate this downtime and be able to serve your customers, hence reducing your risk of impact. You should strive to build a highly available Web architecture, as shown in **Figure 2**.

High Availability at the Web Application Layer

Azure Web Sites creates a stateless cloud solution for your Web application. You need to consider the following points when designing for high availability at the application level:

1. Use Standard mode for your Web sites, and configure it to use at least two Web site instances.
2. Based on your traffic patterns, simulate loads for testing through various tools like Visual Studio and Apache JMeter (jmeter.apache.org) to identify the instance size and how many instances would be needed for your Web site to manage actual traffic levels.
3. Ensure user and session data are stored on a centralized system such as a database or a distributed cache layer. In Azure Web Sites, the File server is shared across all VMs when running in Standard mode.
4. Replicate the Web application layer in at least two regions supported by Azure Web Sites.
5. User content such as media and documents that your Web site uploads or manages should be stored in an Azure Storage account. Make sure the storage account is in the same geographic region as the Web site to reduce latency.
6. Always follow secure coding practices to make your application resilient from malicious attacks.

High Availability at the Database Layer

Data is what truly generates value for any application, so you have to manage it efficiently. For your application to function properly, it's critical to avoid a single point of failure at the database layer. Azure Web site supports the Azure SQL Database and ClearDB MySQL service. Both provide options from low-range to premium solutions.

Azure SQL Database Premium offers more predictable performance and greater capacity for cloud applications using Azure Web sites. It dedicates a fixed amount of reserve capacity for a database including its built-in database replication features. Reserved capacity is ideal for:

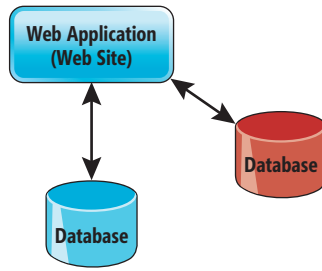


Figure1 Standard Architecture for a Single Web Site

- **High Peak Load:** An application that requires a lot of CPU, memory or I/O to complete its operations is a good candidate for using a Premium database.
- **Many Concurrent Requests:** Some database applications service many concurrent requests. The normal Web and Business editions of Azure SQL Database have a limit of 180 concurrent requests. Applications requiring more connections should use a Premium database with an appropriate reservation size to handle the maximum number of requests.

- **Predictable Latency:** Some applications need to guarantee a response from the database in minimal time. If a given stored procedure is called as part of a broader customer operation, there might be a requirement to return from that call in no more than 20 ms 99 percent of the time. This kind of application will benefit from a Premium database to ensure that computing power is available.

ClearDB offers high-availability SQL routers (or CDBRs) that are custom-built, intelligent traffic managers that monitor these database clusters. When a database node is unhealthy, CDBR automatically redirects to your secondary master. This helps ensure uptime for your database and, in turn, the Web application. When building this design, keep in mind there are recommended pairings of regions ClearDB supports for such scenarios, such as:

1. If you choose one database in the eastern United States, the paired database should be in the western United States.
2. If you choose one database in Northern Europe, the paired database should be in Southern Europe.

High Availability Across Regions

Currently, Azure Web Sites supports multiple regions. It's working on expanding its infrastructure, as well. Architectures that use multiple regions can be classified into active-active Web sites and active-passive Web sites.

Active-Active Web sites: In an active-active Web architecture, you'll have multiple Web sites across regions serving the same application. In this case, traffic is managed across all Web sites, hence they're all considered active.

Active-Passive Web sites: In an active-passive architecture, you'll have a single Web site that will act as a primary Web site. This will serve up the content for all customer traffic. During a failure on this site, customers will be redirected to another site configured and in sync with the primary Web site in a different datacenter mitigating the failure.

In designing your architecture to operate across multiple regions, there are a few challenges you need to consider:

- **Data Synchronization:** This refers to the ability to make a real-time copy of the database across the regions. Any complex system will have a database, file server, external storage and cache—just to name a few components. When designing your architecture, you need to ensure data replicated across multiple regions is in sync to keep your Web application from breaking. This may require some changes

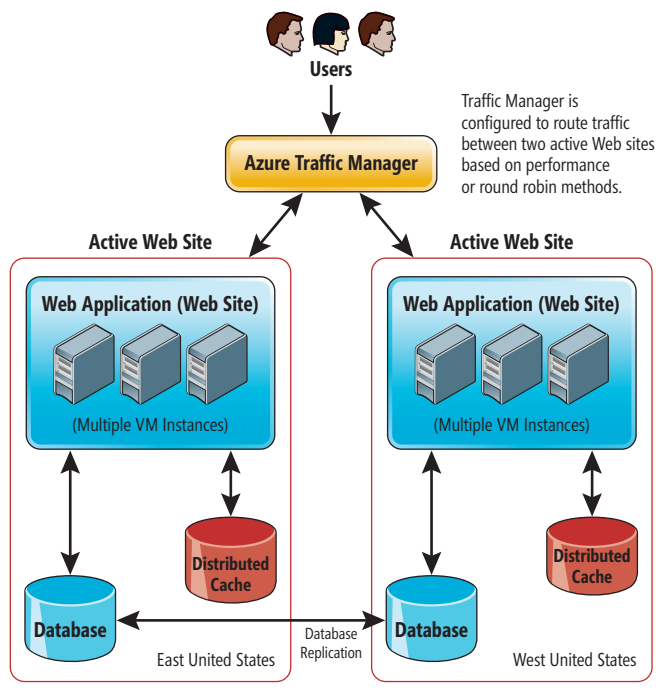


Figure 2 Example of a Highly Available Web Architecture

to your application code to support this scenario. Azure Web Sites supports running a background process called Web Jobs, which lets you build custom tools to manage data synchronization.

- **Network Flow:** This is the ability to manage network traffic across multiple regions. Azure Traffic Manager lets you load balance incoming traffic across multiple hosted Web sites whether they're running in the same data-center or across different datacenters. By effectively managing traffic, you can ensure high performance, availability and resiliency for your applications.

You can use Traffic Manager to ensure high availability and responsiveness for your applications. Traffic Manager provides three load balancing methods:

1. **Failover:** Use this method when you want to use a primary endpoint for all your traffic, but provide one or more backup endpoints in case a primary or backup endpoint becomes unavailable.
2. **Round Robin:** With this method, you can distribute traffic equally across a set of endpoints in the same datacenter or across different datacenters.
3. **Performance:** This method lets requesting clients/users use the closest endpoint to reduce latency by providing endpoints in different geographic locations.

High Availability for Cache Layer

To improve the performance of your Web sites, the caching layer is critical. When handling data with your application, you need to understand how caching may affect both your data and application. When choosing your cache layer, consider the need to maintain consistency of data stored across cache instances. To avoid data

inconsistencies, you should use a distributed caching mechanism. A distributed cache may use multiple servers, so it's scalable and stores application data residing in the database by reducing the number of calls made to the database.

There are many cloud-based caching solutions you can integrate with Azure Web Sites, such as Azure Cache and Memcached Cloud. These are available through the Azure Store in the Azure Management Portal.

Performance Monitoring

Now that you've figured out high-availability strategies for your Web sites, you'll need to evaluate your monitoring options. Azure Web Sites provides two types of monitoring in the Azure Management Portal:

1. **Monitoring with Web site Metrics:** Each Web site dashboard has a Monitor page. This provides performance statistics for each site, such as CPU usage, number of requests, data sent by Web site and so on.
2. **Endpoint Monitoring:** Endpoint monitoring lets you configure Web tests from geo-distributed locations that test response time and uptime for Web URLs. Each configured location runs a test every five minutes. Uptime is monitored with HTTP response codes. Response time is measured in milliseconds. Uptime is considered 100 percent when the response time is less than 30 seconds and the HTTP status code is less than 400. Uptime is 0 percent when the response time is greater than 30 seconds or the HTTP status code is greater than 400. After you configure endpoint monitoring, you can drill down into the individual endpoints to view details of response time and uptime status over the monitoring interval from each of the test locations.

You can perform more detailed and flexible monitoring with the Azure Preview portal. Here, you can build a full-blown DevOps dashboard. **Figure 3** shows how a DevOps dashboard looks when running on Azure Web Sites.

When developing for the cloud, application planning, development and testing is usually conducted elsewhere, such as Visual Studio Online. Monitoring the health of those applications and troubleshooting problems might be done in yet another portal, such as with Application Insights. Billing is displayed on a separate page. Notice a pattern here?

This new portal brings all of the cloud resources, team members, and lifecycle stages of your application together. It gives you a centralized place to plan, develop, test, provision, deploy, scale and monitor those applications. This approach can help teams embrace a DevOps culture by bringing both development and operations capabilities and perspectives together in a meaningful way. You can learn more about this from the blog post, "Building Your Dream DevOps Dashboard with the New Azure Preview Portal," at bit.ly/1sYNRtK.

Recovery Options

Sometimes bad things happen to good applications. Azure Web Sites can help you automatically recover from application failures. Typically, when your monitoring system detects an issue, it alerts

WPF lives!



➔ **XCEED Business Suite for WPF**

The essential set of WPF controls for all your line-of-business solutions. Includes the industry-leading **Xceed DataGrid for WPF**.
A total of 85 tools!

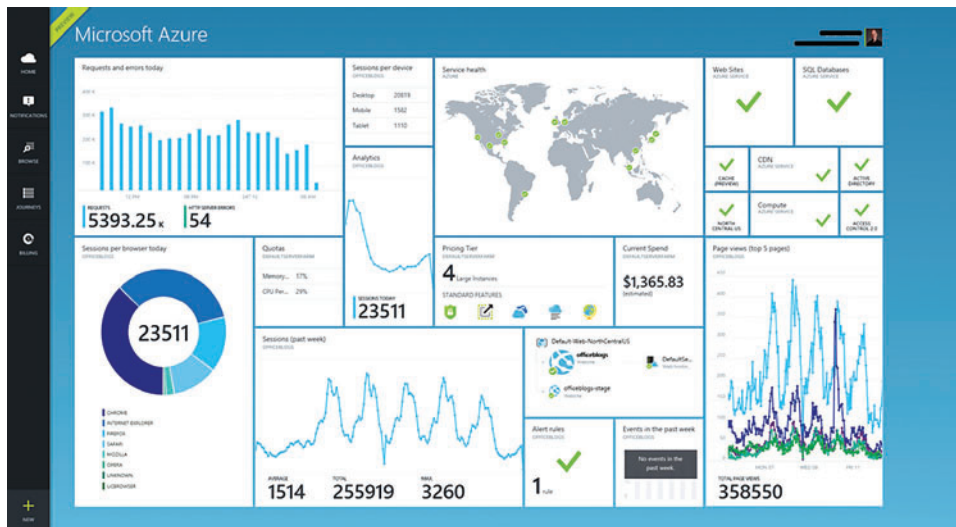


Figure 3 A Typical DevOps Dashboard

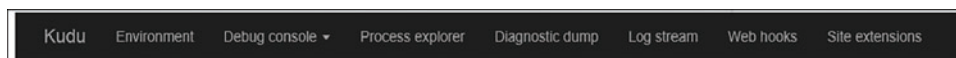


Figure 4 The Kudu Console Provides Diagnostic Tools

an Ops guy in some fashion. The Ops guy then goes ahead and restarts the Web site to get things up and running again.

You can configure the web.config file in your application to detect and act on situations. For example, if X number of requests take Y amount of time to execute in Z amount of time, then perform action ABC (which could be restart the Web site, log an event or run a custom action). Another example may be if my process takes X amount of memory, then perform action ABC. Learn more about recovery procedures from the Microsoft Azure Blog at bit.ly/LOSEvS.

While geo-redundant architectures provide protection from infrastructure-related failures, features such as automated recovery help build application-layer resiliency. In the world of cloud computing, customers are always keen on recovering first compared to performing full-blown diagnostics while the Web site is down. On the other hand, there are other situations and scenarios where diagnostics are essential.

Diagnostic Tools

Diagnostics is similar to peeling away a bad onion. You never know how many layers you need to peel. Azure Web Sites is a fully managed, multi-tenant Platform as a Service (PaaS) offering and as such, does not support remote diagnostics in the VM. This brings a new set of challenges. Here's a list of available tools and diagnostics scenarios for which they're useful.

By default, this service provides various kinds of logging that helps aid troubleshooting. Some of the logs include Web Server Logging, Detailed Error Logging, Application Logging, Failed Request Tracing and so on. Learn more about logging services at bit.ly/1i0MSou.

The Kudu console is another diagnostic tool. This our multi-purpose software configuration management endpoint customers will frequently use for diagnostics purposes. Upon logging onto the Kudu endpoint, you'll see different diagnostic options in the top ribbon (see Figure 4).

The Environment tab gives you a read-only view of things like System Info, App Settings, Connection Strings, Environment variables, System Paths, HTTP headers, and Server Variables specific to your Web site and VMs. This always works well as an aid to ongoing investigation with different data points.

The Debug Console tab gives you CMD- or Windows PowerShell-based remote execution console and file browser access to your sites. You can use the console to perform most standard console operations and arbitrary external commands, like Git commands, navigate the folder UI, download files and folder, upload files and folder using drag and drop, view and edit text files, and so on. Learn more about this by watching the YouTube

video, "The Azure Web Sites Diagnostic Console," at bit.ly/1h0ZZoR.

The Process explorer tab gives you a list of currently active processes your application is able to see and communicate within the Azure Web Sites sandbox. This view will provide you information about process memory and CPU usage. You can also look at detailed information about a given process, generate and download full memory dumps for offline diagnostics, and so on.

The Diagnostic dump tab will give you a memory dump in a .zip file to download. You can then use the dump analyzer like DebugDiag to quickly analyze the memory dump and get some prescriptive guidance.

The Log stream tab lets you live stream HTTP or application logs directly to the console. This is useful for troubleshooting an active issue. Learn more about this by reading Scott Hanselman's blog post, "Streaming Diagnostics Trace Logging from the Azure Command Line (plus Glimpse!)," at bit.ly/1jXwy7q.

This isn't a comprehensive list of debugging tools by any means. Microsoft plans on building more sophisticated diagnostics, where some of the most commonly used scenarios will require only a single click. Until then, enjoy building good applications that are resistant to failure. ■

APURVA JOSHI is a senior program manager with Microsoft, working on the Azure Web Sites team. He joined Microsoft in 2002 and has worked on various Web technologies like IIS and ASP.NET.

SUNITHA MUTHUKRISHNA is a program manager at Microsoft, working on the Azure Web Sites team. She joined Microsoft in 2011 and has been working on Windows Web Application Gallery and Azure Web Sites. Previously, she contributed to IT projects for a couple of non-profit organizations such as the Community Empowerment network and the Institute of Systems Biology, both located in Seattle, Wash.

THANKS to the following technical experts for reviewing this article:
Microsoft Azure Production Management Team



Extreme Performance Linear Scalability

For .NET & Java Apps

(Microsoft Azure Supported)

Cache data, reduce expensive database trips, and scale your apps to extreme transaction processing (XTP) with NCache.

Enterprise Distributed Cache

- Extremely fast & linearly scalable with 100% uptime
- Mirrored, Replicated, Partitioned, and Client Cache
- NHibernate & Entity Framework Second Level Cache

ASP.NET Optimization in Web Farms

- ASP.NET Session State storage
- ASP.NET View State cache
- ASP.NET Output Cache provider

Runtime Data Sharing

- Powerful event notifications for pub/sub data sharing



Download a **FREE** trial!

sales@alachisoft.com

US: +1 (925) 236 3830

www.alachisoft.com

Building a Node.js and MongoDB Web Service

Tejaswi Redkar

The cloud is neutral to all languages. In the cloud, it shouldn't matter whether the apps you run are Node.js, ASP.NET, Java or PHP, because the cloud provides a ready-made infrastructure for running them. The religion of the cloud is agility, and agility allows you to implement new ideas quickly and sunset the ones that don't work in the marketplace. In recent years, Node.js has gained popularity among a new class of developers who favor the asynchronous programming paradigm. Node.js is a JavaScript runtime engine for building client and server applications. It includes an asynchronous I/O framework that helps it handle thousands of concurrent connections on a single thread with minimal CPU or memory overhead. It's a popular myth that Node.js is a Web development framework; in fact, it's a single-threaded runtime engine for running JavaScript code. It has a module-based architecture that allows anyone to build and publish a module to handle specific tasks, such as Web development or accessing a MongoDB database. For example, the Express template engine is a Web application framework you can download as a module in Node.js. The core Node.js engine is built in C++ on top of the Google V8

JavaScript engine, which was designed for the Chrome browser. Node.js is supported in Microsoft Azure Web Sites, and Microsoft also provides development tools and a native SDK for programming with Azure APIs. There's a dedicated developer center for Node.js on the Azure Portal at bit.ly/1iupElQ.

In this article, I'll show you how to develop a RESTful Web service in Node.js that accesses a MongoDB database in the cloud, and how to deploy it to Azure Web Sites.

Node.js is based on an asynchronous development model, which means every method call you make requires a callback to receive the response. Though .NET developers traditionally prefer synchronous (request-response) method calls, asynchronous capabilities have always existed in the Microsoft .NET Framework. With the inclusion of the new `async-await` programming model in the .NET Framework, asynchronous applications have become the norm across Web and mobile applications. In asynchronous programming, the calling function subscribes to the callback event and provides a delegate to process the response. The callback function is called when the processing is complete. It's like an e-mail compared to a phone call.

This article discusses:

- The composition and capabilities of Node.js
- An overview of MongoDB
- Creating and implementing a Node.js Web service that accesses a MongoDB database in the cloud

Technologies discussed:

Microsoft Azure, Microsoft .NET Framework, Node.js, MongoDB

Code download available at:

github.com/dynamicdeploy/appsforazure

DATA ACCESS FOR HIGHLY SCALABLE SOLUTIONS

Master the principles at the foundation of SQL and NoSQL databases and learn to use polyglot persistence to create applications and services that can take full advantage of both.

bit.ly/1d1Lmfs

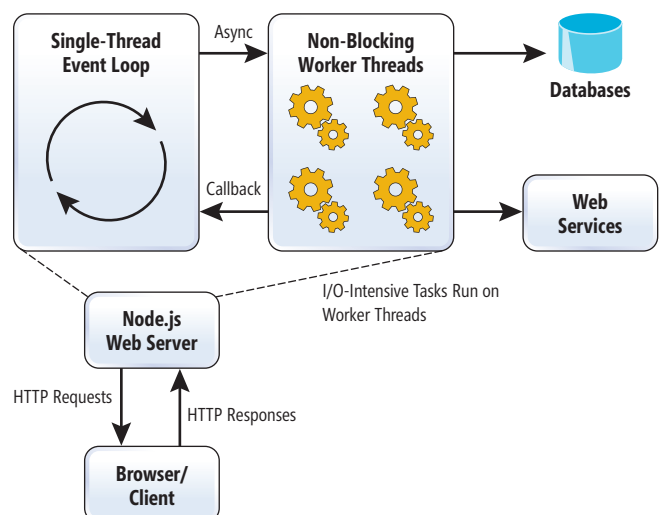


Figure 1 Node.js Request-Response Orchestration

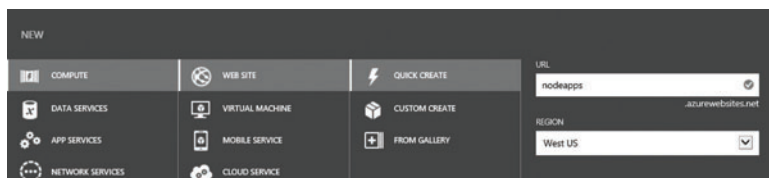


Figure 2 Create a New Azure Web Site

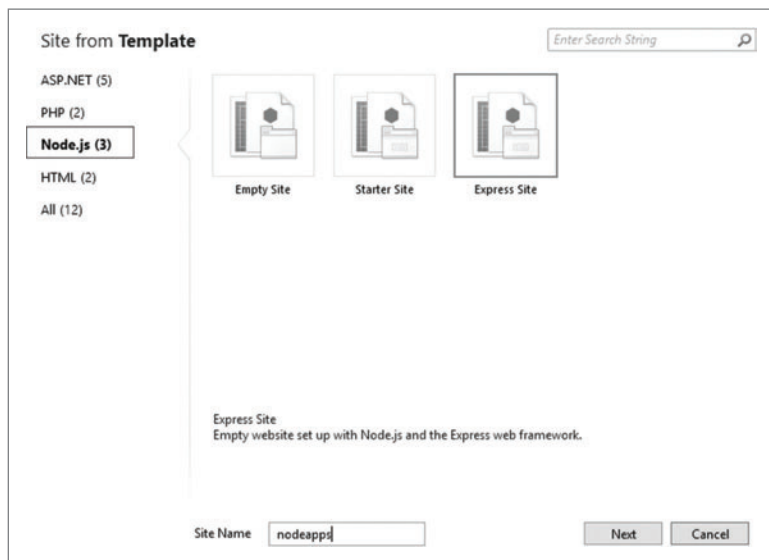


Figure 3 Express Site Template

The following shows a simple Node.js Web server that returns a string when called:

```
var http = require("http");
http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type":
    "text/plain"});
  response.write("Hello MSDN");
  response.end();
}).listen(8080);
```

Note that the response function is called when an HTTP request event is fired. The require function is used for loading modules, similar to loading namespaces from assemblies in the .NET Framework.

There have been a lot of discussions and religious arguments on ASP.NET versus Node.js, but I'm not going to touch that topic in this article. My approach has always been to use the best tool for the job, and with what you're good at do your best.

Node.js Request Processing

As **Figure 1** shows, the Node.js engine starts a single thread for handling concurrent client connections. As the single thread is already initialized, there's no initialization overhead required for processing any rise in requests because the thread quickly delegates the request asynchronously to a worker thread for processing.

If the HTTP request includes a long-running or I/O-intensive task, such as database access or a Web service call, it's executed asynchronously in

non-blocking worker threads. Once the long-running task is complete, the worker threads return the results as a callback to the main thread. The main thread then returns the result back to the client. An important concept to understand here is that the single receiving thread is always available to receive requests and doesn't remain busy with processing because processing is delegated to the worker threads.

Node.js Core Components

Node.js consists of two core components:

Core/Kernel: The kernel of Node.js is written in C++ on top of the Google V8 JavaScript engine. The core itself is single-threaded and is capable of load balancing among CPUs. Node.js is open source and you can get the source code from github.com/joyent/node.

Modules: Modules are similar to NuGet packages in the .NET Framework. The Node.js Package Manager (NPM) is the tool for managing Node.js packages in your development environment. Modules spawn new processes or threads depending on the I/O intensity of the task. Among the popular modules are HTTP, MongoDB, Express (a Web template framework) and Socket.IO. For a list of popular modules, please visit nodejsmodules.org.

Installing and Running Node.js Locally

Before running any Node.js Web site in the cloud, I recommend trying it out locally to get comfortable with the platform. You can install and run Node.js on a Windows platform in just three steps:

1. **Download and install Node.js:** The Node.js Windows installer can be downloaded and installed from nodejs.org/#download. The installer installs the Node.js runtime and NPM.

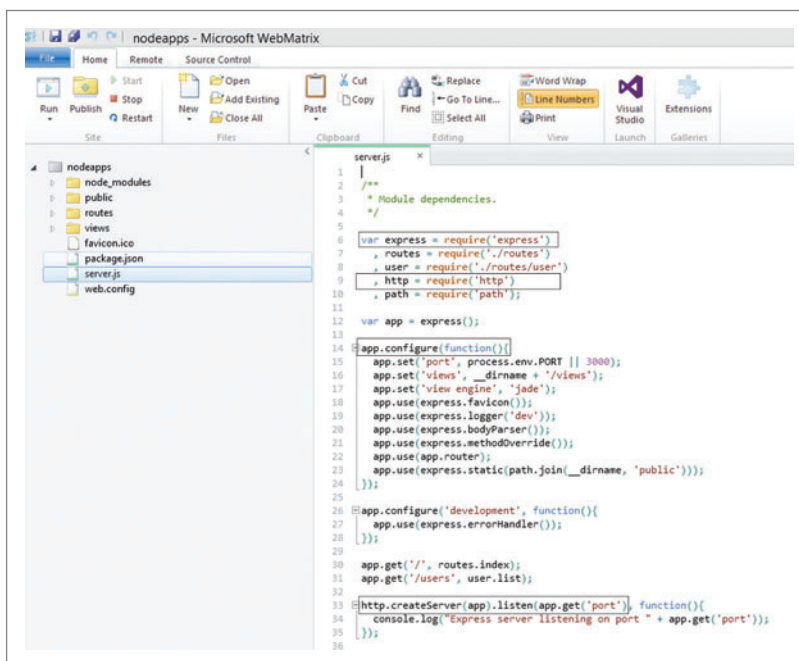


Figure 4 Express File Structure

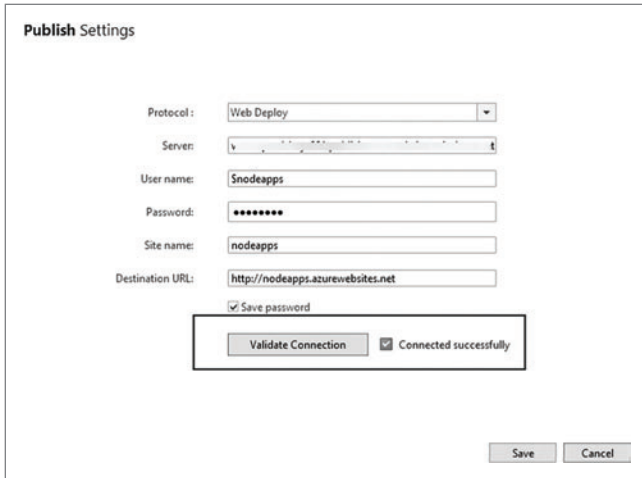


Figure 5 Publish to Microsoft Azure

2. **Create a simple server:** To create an HTTP Server in Node.js, open your favorite text editor, copy the code from **Figure 1**, and save the file as `webserver.js`. The code shown earlier creates a minimalist Web server that listens on port 8080 and responds with the same string for every request.
3. **Run the HTTP server:** Finally, to run the server, open command prompt, navigate to the folder where you saved the `webserver.js` file and type the following:

```
>"C:\Program Files\nodejs\node.exe" webserver.js
```

This command starts the Web server, and you can test it by navigating your browser to `http://localhost:8080`. This simple server should give you enough confidence to try out Node.js as an option for developing Web sites.

Running Node.js on Azure Web Sites

When I first used Node.js, I decided to avoid the command line as much as possible. Having lived my life in Visual Studio, I really value the productivity you can achieve using an IDE. Fortunately, Microsoft has invested in WebMatrix, a powerful tool for developing Node.js applications on Windows. The Visual Studio team has also released the Node.js tools for Visual Studio (nodejstools.codeplex.com). For the rest of this article, I'll use WebMatrix as my primary development tool. From WebMatrix, you can install NPM packages, publish Web sites to Azure, and also run them locally. WebMatrix also installs IISNode for IIS Express, which allows you to host the Node.js application on IIS. You can get more details about IISNode at github.com/tjanczuk/iisnode.

Before building a complete RESTful Web service, I'll show you how to publish a simple Web site to Azure from WebMatrix.

Create a New Azure Web Site You can create a new Web site in Azure from the Azure Portal, as illustrated in **Figure 2**.

The Portal will create a new site with a unique name in the region you specify. The region is important for co-locating your Web site, databases and other services in the same datacenter. Data that leaves a datacenter is charged.

Create an Express Web Site Express is a Web application framework for Node.js. It follows the Model-View-Controller (MVC) pattern and, therefore, allows you to establish routes for building

Node.js MVC Web sites as well as RESTful Web services. You can download Express from expressjs.com.

If you love .NET development in Visual Studio, I recommend mastering WebMatrix for developing Node.js applications. WebMatrix 3 includes a useful, prebuilt Express template. Open it and click on New | Template Gallery. Then, under the Node.js category, select the Express Site template, as shown in **Figure 3**.

Specify a site name and click Next to install IISNode and the Express Site template.

The Node.js server I built earlier won't run as is on Azure Web Sites because the Azure Web Sites infrastructure relies on IIS for running Web sites. Therefore, to run a Node.js Web site, I need the integration between IIS and Node.js that IISNode provides.

Figure 4 illustrates the file structure created by the Express template, and the source code for `server.js`.

Note that the `express` and `routes` modules get imported automatically. I'll use these modules to build the REST services.

Running the Web Site Locally Following my earlier recommendation, click the Run button on WebMatrix to test the Web site on your local machine. If the installation was successful, you should see the Express homepage.

Express template also installs the Jade template engine. Jade is an HTML template engine and is used for building the views generated from the Express framework. Jade is to Express what Razor is to ASP.NET MVC. Therefore, the `index.html` contents are rendered from `/views/index.jade` and `/routes/index.js`. These routes are set up on lines 16, 17 and 30 of `server.js`, as shown in **Figure 4**. For more information on the Jade template engine, visit jade-lang.com.

Publishing the Web site to Azure Now that you've built a Node.js Web site in Express locally, let's publish it to Azure. Click on the Publish button in WebMatrix to start the process. Import the publish profile for your Web site and follow the publishing wizard as shown in **Figure 5**.

If everything goes well, WebMatrix will load your Web site in Internet Explorer.

The URL in the browser should be the Azure Web Site to which you published the Web site. This is a good demonstration of how easy it is to develop and deploy Node.js Web sites and Web services to Azure. Now, I'll take a moment to look at MongoDB. I'll come back to Node.js later to build Web services.

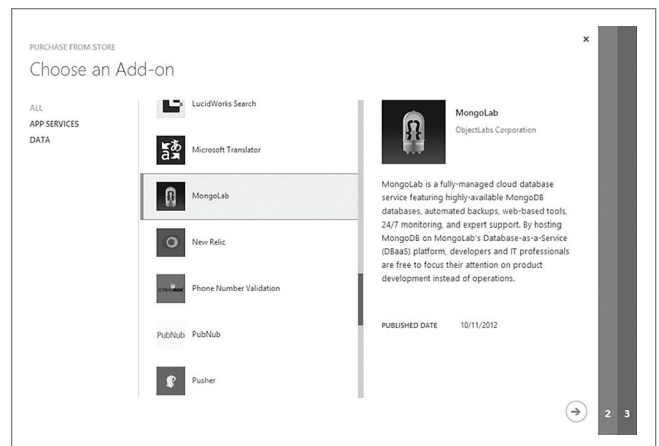


Figure 6 MongoDB Add-on from MongoLab

GdPicture.NET 10



- Document viewing, processing, printing and scanning (TWAIN & WIA).
- Reading, writing and converting vector and raster images in more than 90 formats, PDF included.
- OMR, OCR, barcode reading and writing (linear & 2D).
- Annotations for image and PDF within Windows and Web applications.
- Color detection engine for image and PDF compression.

And much more ... 

All-In-One Document Imaging SDK

Royalty-Free Document Imaging Toolkits
for .NET and COM/ActiveX



GdPicture.NET 10 Plugins



Color detection



DICOM image reader



MICR reader

- Full managed PDF support
- Full annotations support for PDF and images
- OCR
- Forms processing
- JBIG2 encoding
- 1D and 2D barcode reading and writing



Try GdPicture.NET 10
FREE for 30 days

www.gdpicture.com

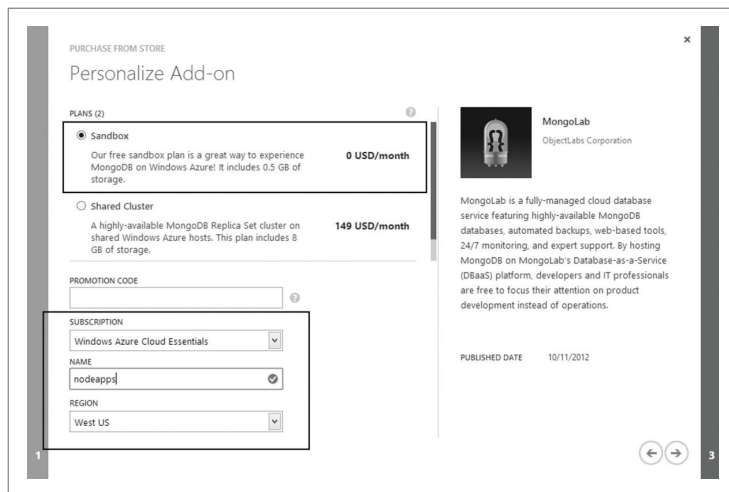


Figure 7 Configuring MongoDB

MongoDB Overview

MongoDB is an open source, scalable, high-performance, document-oriented database that includes most of the infrastructure capabilities of relational databases, such as replication, sharding, indexing and failover. MongoDB also offers auto-sharding and has built-in Map-Reduce processing capabilities, which aren't offered by most relational databases today. Relational databases were designed in an era where storage systems were expensive. Storing data in a relational format allowed developers to optimize storage space without compromising quick data retrieval. In today's world, storage is cheap when compared to compute costs. In the cloud, storage prices continue to decline. As a result, saving and retrieving data in relational format is expensive compared with storing it in the cloud. The demand for continuous data delivery is on the rise, with more applications expecting real-time data from your data storage. MongoDB allows you to save objects and documents without breaking them down into relational components. This reduces the processing load on the application, as well as the database.

MongoDB isn't recommended for applications that require deep object relationships because it isn't designed to maintain and retrieve object and data relationship linking, as relational databases are. If your application requires deep relations and SQL for retrieval, then use a relational database. If your application requires fast object storage and retrieval, use MongoDB. As I suggested earlier, use the best tool for the job.

Provisioning MongoDB in Azure

MongoDB is available as an add-on in the Azure Store and you can install a sandbox version for free. Log in to your Azure Portal and install MongoDB (MongoLab) from the New | Store | Add-on menu, as shown in Figure 6.

On the Personalize Add-on page, select the Sandbox version, and be sure to install the MongoDB instance in the same region as your Web site, as shown in Figure 7.

Once the installation is complete, navigate to the add-on page and save the

connection string for the installed database, as shown in Figure 8. You'll use the connection string to connect from your Node.js Web service.

MongoDB is now running in Azure. MongoLab (by ObjectLabs Corp.) has its own dedicated Management Portal you can view by clicking on the Manage Your Add-on link.

You now have a MongoDB database running in the cloud, and you've created an empty Node.js template. To complete the application, you need to populate the MongoDB database, and create the Web services that retrieve data from this database. This is standard software development in the cloud.

Installing Node.js Prerequisites for MongoDB Connectivity

JSON is a first-class citizen in MongoDB and, therefore, complements Node.js. Because the Web service will connect to MongoDB, I need to install the latest Node.js driver for MongoDB from the NPM Gallery (see Figure 9).

The MongoDB connection information gets stored in a configuration file. The nconf module allows you to read configuration information from files (see Figure 10).

Once the mongodb and nconf modules are installed, you've completed all the prerequisites for setting up a Node.js and MongoDB development environment.

Creating the REST Web Service Signatures

To create a REST Web service, you first define dependencies and routes in server.js:

```
var express = require('express')
, routes = require('./routes')
, user = require('./routes/user')
, http = require('http')
, path = require('path'),
  pkgs=require('./routes/pkgs');
```

Next, you define the REST functions in the /routes/pkgs.js file because, in the preceding code, you delegate the function handling to the pkgs module. In the routes folder, create a file named pkgs.js that defines the Web service operations, as shown in Figure 11.

To test the basic functioning of the operations, just create function skeletons returning static data. You can add the database access code later. In server.js, define the route methods that correspond to the module methods you just created:

```
app.get('/pkgs', pkgs.findAll);
app.get('/pkgs/:id', pkgs.findById);
app.post('/pkgs', pkgs.addPkg);
app.put('/pkgs/:id', pkgs.updatePkg);
```

If you're familiar with the ASP.NET Web API, you'll quickly understand these routes and their mapping to the appropriate implementation class. With the route declaration and mapping,

any request to /pkgs will be routed to one of the functions in pkgs.js. With the Web service operation signatures and definitions in place, you can test if the routes are working properly by running the Web site locally, as well as in Azure. The URLs to test are: http://[Web site URL]/pkgs, http://[Web site URL]/pkgs/

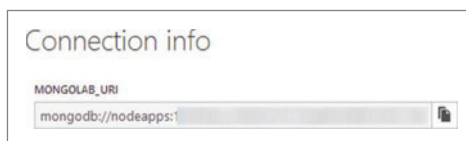


Figure 8 Connection String for the Installed Database

DEVELOPED FOR INTUITIVE USE

DynamicPDF—Comprehensive PDF Solutions for .NET Developers

ceTe Software's DynamicPDF products provide real-time PDF generation, manipulation, conversion, printing, viewing, and much more. Providing the best of both worlds, the object models are extremely flexible but still supply the rich features you need as a developer. Reliable and efficient, the high-performance software is easy to learn and use. If you do encounter a question with any of our components, simply contact ceTe Software's readily available, industry-leading support team.



DynamicPDF

[WWW.DYNAMICPDF.COM](http://www.DynamicPDF.com)



TRY OUR PDF SOLUTIONS FREE TODAY!

www.DynamicPDF.com/eval or call 800.631.5006 | +1 410.772.8620

ceTe software

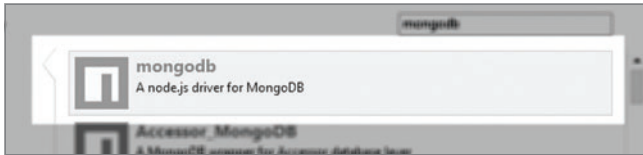


Figure 9 Node.js Driver for MongoDB

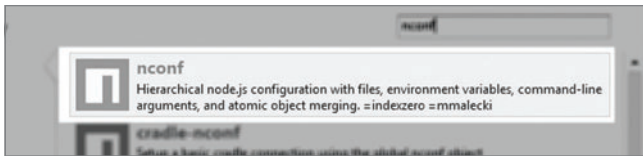


Figure 10 The nconf Module

remote, and `http://[Web site Url]/pkgs/1`. If these URLs return the expected values, the routes are working fine and you can proceed with the MongoDB integration.

Implementing REST Web Service Operations

Now you need to implement the Web service operations to connect to MongoDB and retrieve the data. Four steps will accomplish this:

1. Creating a config.json file and storing the MongoDB connection string in it.
2. Initializing the MongoDB client.
3. Populating MongoDB with sample data. In this step, the function makes an HTTP GET call to `http://storage.azure.com/appsforazureobjecttest/servicepackages.json` and stores the retrieved data into the MongoDB instance.
4. Implementing the data access functions for retrieving and storing data in MongoDB.

Figure 11 REST Operation Signatures

```
exports.findAll = function(req, res) {
  res.send([ {name: 'app1'}, {name: 'app2'}, {name: 'app3'} ]);
};
exports.findById = function(req, res) {
  res.send({id: req.params.id, name: "DisplayName", description: "description"});
};
exports.addPkg = function (req, res) {
  res.send("Success");
};
exports.updatePkg = function (req, res) {
  res.send("Success");
};
```

Figure 12 Connect to MongoDB Database

```
var mongo = require('mongodb');
var MongoClient = mongo.MongoClient

MongoClient.connect(connectionString, function (err, db) {
  if (err) throw err;

  if (!err) {
    console.log("Connected to 'pkgsdb' database");
    db.collection('pkgs', { strict: true }, function (err, collection) {
      if (err) {
        console.log(
          "The 'pkgsdb' collection doesn't exist.
          Creating it with sample data...");
        populateDB(db);
      }
    });
  }
});
```

Creating the config.json File The config.json file, which is similar to a .NET app.config but in JSON format, stores your configuration information. Add the following name-value pair to the new file:

```
{
  "MONGODB_URI" :
    "mongodb://nodeapps:xxxxxxx.f.migd9GGB9Ck3M17j1kVCUVI-@ds027758.
    mongolab.com:27758/nodeapps"
}
```

The value is the MongoDB connection string you saved earlier from the Azure Portal.

In `pkgs.js`, you have to import the `nconf` module to read the configuration values:

```
var nconf = require('nconf');
nconf.env().file({ file: 'config.json' });
var connectionString = nconf.get("MONGODB_URI");
```

The `connectionString` variable can then be used by the functions to connect to the MongoDB database. Note that `connectionString` is declared as a global variable in `pkgs.js` so that all the functions can access it.

Initializing the MongoDB client: To initialize a connection to the MongoDB database, you have to import the `mongodb` module and call the `connect` method, as shown in **Figure 12**.

When the connection is successful, the `db` variable in **Figure 12** contains the database connection object. The `db.collection` function tries to connect to a collection named `pkgs`. If `pkgs` doesn't exist, a new one is created by calling the `populateDB` function. In a real-world application, you wouldn't need to do this because the collection would actually be created before running the application. (A collection in MongoDB is a grouping of related documents, analogous to a database table in a relational database system.)

Populating MongoDB with Sample Data For this article, I used sample data available as a collection of software packages from `dynamicdeploy.com`. In the `populateDB` function, I load this collection in JSON format from `http://storage.azure.com/appsforazure-objecttest/servicepackages.json`, as shown in **Figure 13**.

Note that JSON is a first-class citizen in Node.js, so you don't have to import any module to parse the JSON objects. The `collection.insert` function inserts the entire JSON document (var `pkgs`) into the MongoDB collection. MongoDB determines the schema of the objects at run time and makes intelligent decisions about storing them. In a relational database, you'd have to define the schema of the table before storing any data in it. A collection

Figure 13 Populating the Database

```
var populateDB = function (db) {
  var body = "";
  var url =
    "http://storage.azure.com/appsforazureobjecttest/servicepackages.json";
  http.get(url, function (res2) {
    res2.on('data', function (chunk) {
      body += chunk;
    });
    res2.on("end", function () {
      var pkgs = JSON.parse(body);
      db.collection('pkgs', function (err, collection) {
        collection.insert(pkgs, { safe: true }, function (err, result) {});
      });
    });
    res2.on("error", function (error) {
      // You can log here further
    })
  });
};
```

Figure 14 The Find Functions

```

exports.findById = function (req, res) {
  var id = req.params.id;
  console.log('Retrieving pkg: ' + id);

  MongoClient.connect(connectionString, function (err, db) {
    if (err) throw err;

    if (!err) {

      db.collection('pkgs', function (err, collection) {
        collection.findOne({ '_id': new BSON.ObjectId(id) },
          function (err, item) {
            res.send(item);
          });
      });
    }
  });
};

exports.findAll = function (req, res) {

  MongoClient.connect(connectionString, function (err, db) {
    if (err) throw err;

    if (!err) {
      db.collection('pkgs', function (err, collection) {
        collection.find().toArray(function (err, items) {
          res.send(items);
        });
      });
    }
  });
};

```

gives you the flexibility to modify the schema at run time just by changing the properties of the JSON objects. If the object properties change, MongoDB automatically applies those changes to the schema before storing the object. This is useful for building applications, such as social and the Internet of Things feeds, which dynamically change with changes in data types.

Implementing the Data Access Functions Finally, you need to implement the data access functions for the HTTP GET, POST and PUT functions declared earlier. The HTTP GET function is mapped to the `findById` and `findAll` functions in the Web service, as shown in **Figure 14**.

The `findById` function retrieves an object by its Id, whereas the `findAll` function retrieves all the objects. Functions `collection.findOne` and `collection.find` retrieve the results of these operations from MongoDB, respectively. Similarly, **Figure 15** shows the `addPkg` method, which is mapped to an HTTP POST, and the `updatePkg` method, which is mapped to an HTTP PUT.

If you've followed the article so far, you should be able to implement a delete function on your own. I'll leave that as an exercise for you. Save and run the Web site locally first and then publish it to Azure Web Sites to test each REST function. The complete source code for the REST Web service and the Web site is available at github.com/dynamicdeploy/appsforazure.

Once you have the end-to-end application running, you can scale out your MongoDB infrastructure (or service) depending on the user load. As this article demonstrates, you don't need any infrastructure to deploy large-scale Node.js and MongoDB Web sites on Azure. The skeleton I provided in the accompanying source code will help you get started building Web services and Web sites in Node.js on Azure Web Sites.

Figure 15 Add and Update Functions

```

exports.addPkg = function (req, res) {
  var pkg = req.body;
  console.log('Adding pkgs: ' + JSON.stringify(pkg));

  MongoClient.connect(connectionString, function (err, db) {
    if (err) throw err;

    if (!err) {
      db.collection('pkgs', function (err, collection) {
        collection.insert(pkg, { safe: true }, function (err, result) {
          if (err) {
            res.send({ 'error': 'An error has occurred' });
          } else {
            console.log('Success: ' + JSON.stringify(result[0]));
            res.send(result[0]);
          }
        });
      });
    }
  });
};

exports.updatePkg = function (req, res) {
  var id = req.params.id;
  var pkg = req.body;
  console.log('Updating pkgs: ' + id);
  console.log(JSON.stringify(pkg));

  MongoClient.connect(connectionString, function (err, db) {
    if (err) throw err;

    if (!err) {
      db.collection('pkgs', function (err, collection) {
        collection.update({ '_id': new BSON.ObjectId(id) },
          pkg, { safe: true }, function (err, result) {
            if (err) {
              console.log('Error updating pkg: ' + err);
              res.send({ 'error': 'An error has occurred' });
            } else {
              console.log('' + result + ' document(s) updated');
              res.send(pkg);
            }
          });
      });
    }
  });
};

```

Wrapping Up

The purpose of this article was to get you comfortable building Web services in Node.js on Azure Web Sites. Node.js and MongoDB complement each other and because both are available as Platform as a Service (PaaS), there's no up-front infrastructure cost for building a complete Web site and scaling it out dynamically. As a developer, you can focus on building and consuming the Web services right from the cloud into your mobile apps. Therefore, I believe Azure Web Sites is PaaS done right. ■

TEJASWI REDKAR is an author and a software developer. He currently works for Microsoft as a director of application platform strategy and communities. His book, "Windows Azure Web Sites: Building Web Apps at a Rapid Pace" (Dynamic Deploy LLC, 2013), is the most-comprehensive and best-selling on the topic. Redkar is also the creator of *appsforazure* Windows Store App and *dynamicdeploy.com*, where he experiences first-hand running production apps in the cloud. You can reach out to him at tejaswi_redkar@hotmail.com and follow him on Twitter at twitter.com/tejaswiredkar.

THANKS to the following technical experts for reviewing this article:
Microsoft Azure Product Management Team

Hybrid Connectivity: Connecting Azure Web Sites to LOB Apps Using PortBridge

Tejaswi Redkar

Contrary to popular belief, not everything is going to the cloud, at least not yet. Just as all cars are not yet hybrid and most appliances are not yet energy efficient, a good amount of software will still be running on-premises for years to come. Deep investments are difficult to abandon, and most companies require a significant business justification for modernizing an existing app. Do you have solar panels in your house? In spite of long-term energy savings and tax benefits, people are reluctant to invest in them due to upfront costs. It's not that different for organizations when it comes to moving to the cloud.

Fortunately, however, hybrid connectivity allows you to modernize your applications by tapping into the power of the cloud, while still plugging into existing software services that run on-premises

in your datacenter. In this article, I'll show you how to build a Web site that leverages the Azure Service Bus for connecting to software services running in non-Azure datacenters. Instead of using the Service Bus API directly, I'll use a popular utility called PortBridge to connect Azure Web Sites with on-premises services.

Hybrid Connectivity in Azure

Azure includes a number of services for building hybrid applications. For hybrid connectivity, the two most commonly used are Azure Virtual Network and Azure Service Bus. Azure Virtual Network lets you extend your on-premises network into Azure, thus carving out a private hybrid network between Azure and the datacenter. The extension between Azure and your datacenter happens at the network layer, rather than the application layer in Service Bus. Because this article is about application connectivity using Service Bus, I won't be covering Azure Virtual Network here. For more information, visit bit.ly/QA0DgX.

The Service Bus Relay service lets you connect two applications residing behind a firewall. The applications can be residing in Azure or your datacenters or both. Service Bus Relay gives you the ability to register your application's Windows Communication Foundation (WCF) endpoints into the Service Bus registry in Azure datacenters. Method invocations can then be securely exchanged between the client and the server within the Service Bus infrastructure and then communicated to the respective applications running in your datacenter. This is ideal, for example, for a scenario in which a utility company needs access to the HVAC or power meter for collecting data and sending important events to

This article discusses:

- Hybrid connectivity in Microsoft Azure
- Using the PortBridge utility to enable hybrid connectivity
- Creating an example hybrid solution
- Performance and security considerations
- Testing and deploying the solution

Technologies discussed:

Microsoft Azure Web Sites, Windows Communication Foundation, PortBridge

Code download available at:

github.com/dynamicdeploy/portbridge

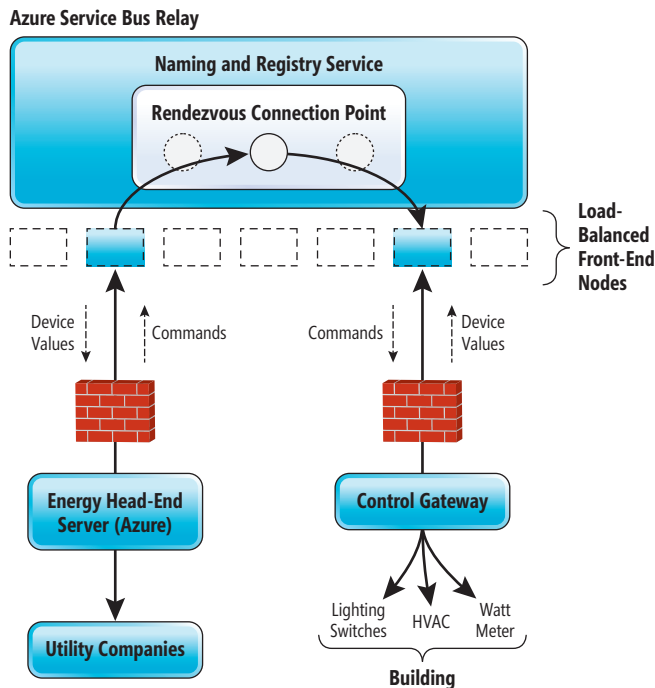


Figure 1 Service Bus Relay Scenario at a Utility Company

these devices from a central server running in Azure. **Figure 1** illustrates this example, where the utility company has a head-end running in Azure and devices in buildings.

The control gateway is a device running in buildings that's responsible for controlling electrical devices. Each control gateway has a WCF endpoint registered with the Service Bus registry with a globally unique identifier. Whenever the head-end service, running in one of the Azure Compute Services, wants to communicate with a particular control gateway, it opens a connection to the globally unique endpoint in Service Bus and starts sending or retrieving messages to or from the control gateway. Typically, the control gateway resides behind the building's firewall. But what if you have a non-WCF service (such as a SQL Server database) running on-premises that you want to connect from an Azure Web Site?

Introducing PortBridge

PortBridge is a utility built on top of the Service Bus API that allows you to communicate between any TCP-based service endpoints running on-premises and on Azure. The core concept of PortBridge and the prototype were developed by Clemens Vasters (bit.ly/SI93GM). I've modified it a little for this article and compiled it with the recent version of the Service Bus SDK. When you build applications for Service Bus Relay, you're forced to build a WCF interface for all the relevant endpoints you want to expose in the cloud. This can get inconvenient and tedious when you have a large number of endpoints or generic platform endpoints, like databases and search engines. Adding another WCF abstraction on top of these services doesn't make sense. Instead, you can use PortBridge to expose non-WCF TCP endpoints for Service Bus Relay connectivity. Front-end applications like Azure Web Sites can then take advantage of connecting to any TCP data source residing in your

datacenter behind a firewall. Conceptually, PortBridge creates a generic WCF interface as shown in **Figure 2**.

With just three generic methods, PortBridge acts as a proxy between the client and server endpoints and forwards all TCP calls to the designated server over a Service Bus Relay. The biggest advantage of using PortBridge is you don't have to build a WCF interface for every on-premises endpoint you want to enable for hybrid connectivity.

PortBridge is composed of two components:

1. An Agent that runs closer to the client application, or in a virtual machine (VM) in the Azure Infrastructure as a Service (IaaS).
2. A Windows Service (or a console application) that runs on-premises and acts as a proxy for service endpoints running in the same environment.

Some common use cases for PortBridge are:

- Connecting to any TCP-based data store on-premises
- Connecting to third-party Web services that can't be migrated to Azure
- Remote desktop connections

In this article, you'll learn how to deploy and configure PortBridge to enable communication between Azure Web Sites and a SQL Server database running on your local machine.

A Hybrid Solution

Azure Web Sites typically forms the Web front end or Web services of your application tier. But in many situations, you can't migrate some of the data sources that power the Web site to Azure. In such cases, PortBridge is an ideal solution for connecting Azure Web Sites to databases running on-premises. Azure Web Sites communicates with the PortBridge Agent, which in turn communicates with the PortBridge service via Service Bus. The PortBridge service then forwards the call to the destination database. **Figure 3** illustrates the architecture of the hybrid Web site I'll build in this article and shows a typical message exchange in a PortBridge (or Service Bus Relay) environment:

1. The on-premises PortBridge console application (or Windows service) registers the database's TCP endpoint

Figure 2 The Generic PortBridge Windows Communication Foundation Interface

```
namespace Microsoft.Samples.ServiceBus.Connections
{
    using System;
    using System.ServiceModel;
    [ServiceContract(Namespace="n:",
        Name="idx", CallbackContract=typeof(IDataExchange),
        SessionMode=SessionMode.Required)]
    public interface IDataExchange
    {
        [OperationContract(Action="c", IsOneWay = true, IsInitiating=true)]
        void Connect(string i);
        [OperationContract(Action = "w", IsOneWay = true)]
        void Write(TransferBuffer d);
        [OperationContract(Action = "d", IsOneWay = true, IsTerminating = true)]
        void Disconnect();
    }

    public interface IDataExchangeChannel : IDataExchange, IClientChannel { }
}
```

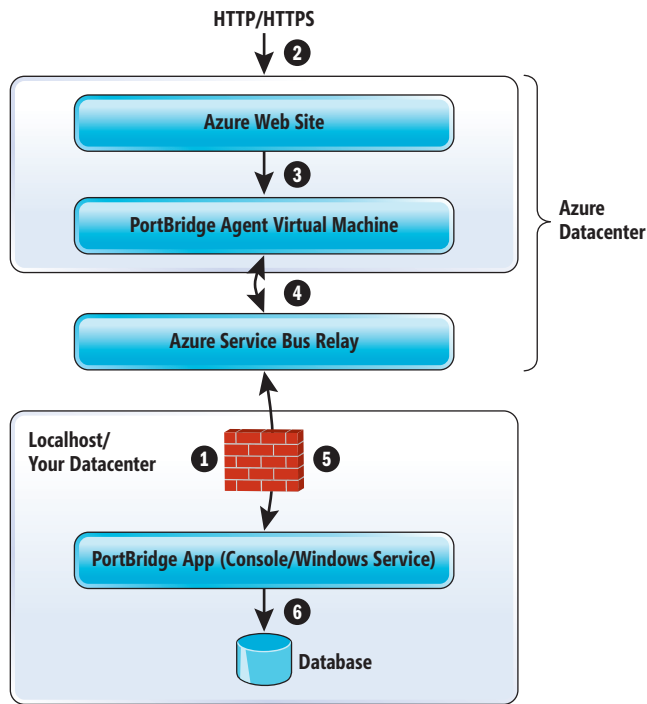


Figure 3 PortBridge Sample Architecture

(1433 for SQL Server) within the Service Bus naming registry with a unique URI. Service Bus Relay opens an outbound bi-directional connection with the console application.

2. When an HTTP or HTTPS request comes in to Azure Web Sites, the Web site opens the database connection as usual, but with the IP address of the PortBridge Agent VM instead.
3. The PortBridge Agent VM acts as the database proxy in the cloud and has two communication interfaces—for receiving TCP requests and for connecting to the PortBridge console's Service Bus Relay endpoint.
4. The PortBridge Agent routes the database request to Service Bus Relay service. The beauty of the solution is that for Azure Web Sites, it's business as usual. The API to the database hasn't changed, only the IP address has.
5. Service Bus Relay routes the call to the PortBridge application running on your localhost (or datacenter).
6. Finally, the call reaches the database, and the data is retrieved using the same connection.

Note that there are a number of components involved in communicating over PortBridge, and you do need an extra VM that might not be needed if using Service Bus directly. As I mentioned earlier, the advantage here is the generalization of exposing any TCP endpoints—you can reuse the same PortBridge Agent VM to connect with multiple data sources running in different locations. Now that I've shown you the system architecture of the solution, I'll start constructing it.

Creating a RESTful Web Service

In order to keep this article simple, I'll design a RESTful ASP.NET Web API ser-

Figure 4 Countries Web Service

```
public class CountriesController : ApiController
{
    private string DatabaseConnectionString { get; set; }

    public CountriesController()
    {
        try
        {
            DatabaseConnectionString =
                ConfigurationManager.
                ConnectionStrings["DatabaseConnectionString"].ConnectionString;
        } catch (Exception ex)
        {
            Trace.TraceError(ex.Message);
        }
    }

    // GET: api/Countries
    public IEnumerable<Country> Get()
    {
        return CountryService.GetCountries(DatabaseConnectionString);
    }

    // GET: api/Countries/AD
    public Country Get(string id)
    {
        return CountryService.GetCountry(DatabaseConnectionString, id);
    }
}
```

vice called Countries that retrieves a list of all countries from a SQL Server database table. In the development process, I always recommend building and testing locally before deploying to the cloud. **Figure 4** shows the code for the Countries Web service.

The Web service has only two methods—Get and Get(string id). The Get method retrieves all the countries from the database table and the Get(id) method retrieves a Country object by the country code. **Figure 5** illustrates the Country database table in the local SQL Server database.

Figure 6 shows the CountryService class that retrieves the data from the database and returns Country objects.

After the database and the Web service are ready, you can quickly test the Web service locally by pressing F5 to run it from Visual Studio. If everything goes well, you have a Web service that works locally. Now you want to move it to the cloud, but you can't migrate the database to the cloud because it's shared with other applications that will still be running on-premises. PortBridge fits perfectly in this scenario, with minimal or no code changes to your Web service.

Setting up PortBridge

Before moving the Web service to the cloud, you need to configure and test PortBridge. Because there are multiple tiers in the architecture, it's important to get the configuration of all the components right at the start. I usually create a table listing input and output endpoints for each component (see **Figure 7**).

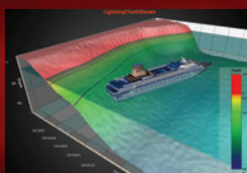
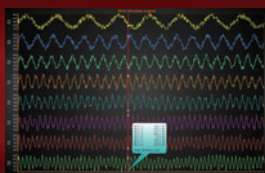
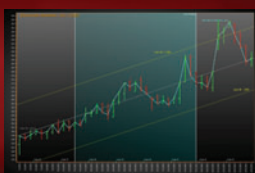
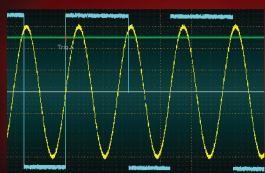
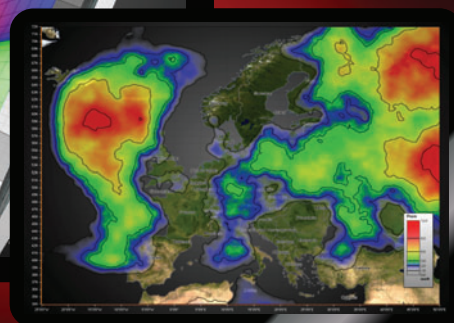
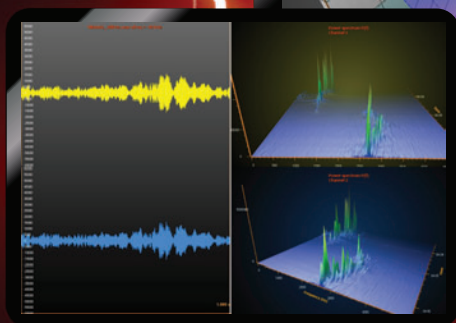
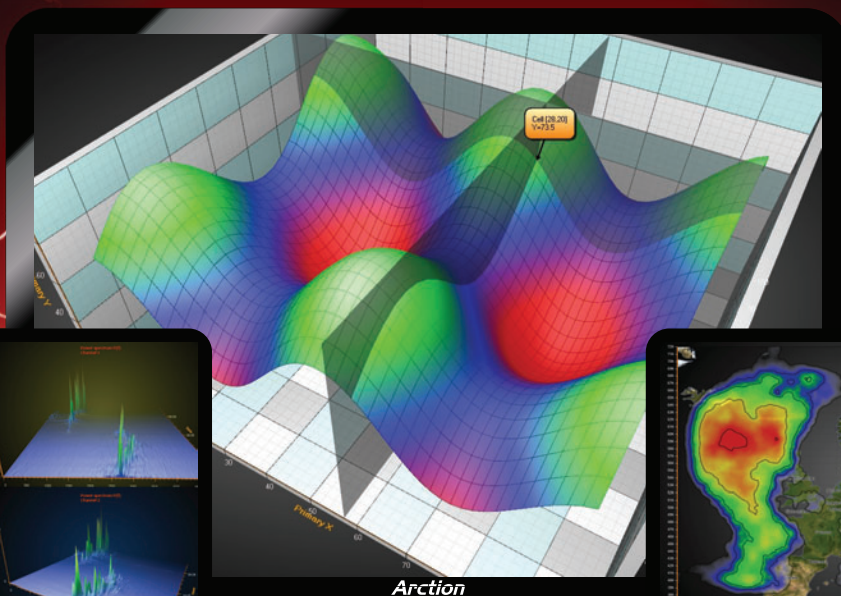
This table might seem trivial in this case, but for applications with hundreds of endpoints, a table like this will help you configure services correctly. As discussed earlier, the PortBridge Agent resides in a VM and has two endpoint interfaces—input from the application and output

Column Name	Data Type	Allow Nulls
CountryID	int	<input type="checkbox"/>
CountryName	nvarchar(255)	<input type="checkbox"/>
CountryCode	nvarchar(255)	<input type="checkbox"/>
		<input type="checkbox"/>

Figure 5 Countries Database Table

The fastest rendering data visualization components
for WPF and WinForms...

LightningChart



HEAVY-DUTY DATA VISUALIZATION TOOLS FOR SCIENCE, ENGINEERING AND TRADING

WPF charts performance comparison

Opening large dataset	LightningChart is up to 977,000 % faster
Real-time monitoring	LightningChart is up to 2,700,000 % faster

Winforms charts performance comparison

Opening large dataset	LightningChart is up to 37,000 % faster
Real-time monitoring	LightningChart is up to 2,300,000 % faster

Results compared to average of other chart controls. See details at www.LightningChart.com/benchmark. LightningChart results apply for Ultimate edition.

- Entirely DirectX GPU accelerated
- Superior 2D and 3D rendering performance
- Optimized for real-time data monitoring
- Touch-enabled operations
- Supports gigantic data sets
- On-line and off-line maps
- Great customer support
- Compatible with Visual Studio 2005...2013



Download a free 30-day evaluation from
www.LightningChart.com

Arction
Pioneers of high-performance data visualization

Figure 6 The CountryService Class

```
public class CountryService
{
    public static IEnumerable<Country> GetCountries(string connectionString)
    {
        IList<Country> countries = new List<Country>();
        using (IDataReader reader = SqlHelper.ExecuteReader(connectionString,
            System.Data.CommandType.Text, "SELECT CountryId, CountryName,
            CountryCode FROM Country"))
        {
            while(reader.Read())
            {
                Country c = new Country()
                {
                    CountryId = Convert.ToInt32(reader["CountryId"]),
                    CountryCode = Convert.ToString(reader["CountryCode"]),
                    CountryName = Convert.ToString(reader["CountryName"]),
                };

                countries.Add(c);
            }
        }

        return countries;
    }

    public static Country GetCountry(string connectionString, string countryCode)
    {
        Country c = new Country();
        using (IDataReader reader = SqlHelper.ExecuteReader(connectionString,
            System.Data.CommandType.Text,
            "SELECT CountryId, CountryName, CountryCode FROM Country WHERE
            CountryCode=@countryCode",
            new SqlParameter("@countryCode", countryCode)))
        {
            if (reader.Read())
            {
                c.CountryId = Convert.ToInt32(reader["CountryId"]);
                c.CountryCode = Convert.ToString(reader["CountryCode"]);
                c.CountryName = Convert.ToString(reader["CountryName"]);
            }
        }

        return c;
    }
}

[DataContract]
public class Country
{
    [DataMember]
    public int CountryId { get; set; }

    [DataMember]
    public string CountryName { get; set; }

    [DataMember]
    public string CountryCode { get; set; }
}
```

to a Service Bus Relay endpoint. Similarly, the PortBridge service running on your local machine has two endpoints—input from Service Bus Relay and output to the SQL Server database. Once you have the configuration parameters noted down, you can start the deployment process.

Creating a Service Bus Namespace

Because the communication backbone of this solution is Service Bus Relay, I have to first create a Service Bus namespace from the Azure portal (manage.windowsazure.com), as shown in **Figure 8**.

The countries namespace in **Figure 8** is globally unique and will be used to create a globally unique endpoint for the services. Once you've created the namespace, click on the Connection Information button at the bottom of the page to see the namespace's access information. You will need the Default Issuer and the Default Key for accessing the namespace, shown in **Figure 9**, so make a note of these.

The PortBridge Agent and PortBridge application both require these credentials for connecting to the namespace.

Configuring the PortBridge Application

The PortBridge application has a custom configuration section:

```
<portBridge serviceBusNamespace="countries" serviceBusIssuerName="owner"
serviceBusIssuerSecret="XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
<hostMappings>

<!-- Open HTTP, SQL Server, RDP and ElasticSearch ports-->
<add targetHost="TREDKAR-W530" allowedPorts="1433" />

</hostMappings>
</portBridge>
```

The namespace, issuer name, and the secret are required to connect to the Service Bus Relay namespace I created earlier. The host-mapping section lists the target service host. In my example, it's the local machine and 1433 port. You can add comma-separated ports in order to externalize multiple endpoints on the same host.

You can also add multiple target hosts as long as they're reachable from the PortBridge application machine. Once the configuration is complete, start the PortBridge application on your local machine. If everything goes as expected, you should see a screen similar to the one in **Figure 10**.

The PortBridge application creates an outbound connection to the Service Bus Relay service on the countries namespace and creates a unique name with the format PortBridge/[target host], where [target host] is the targetHost parameter in the PortBridge application's configuration file, as shown in **Figure 11**.

The PortBridge part of the name is hardcoded, whereas the target host changes depending on the number of hosts you've created. It's important to note that to keep the name within the namespace unique, every target host will have only one entry in the hostMappings section of the configuration file. Also note that if you don't configure the targetHost parameter correctly, the created endpoint will not match the service you want to represent and communications will fail.

For information on outbound ports required by Service Bus Relay bindings, visit bit.ly/1l8lncx.

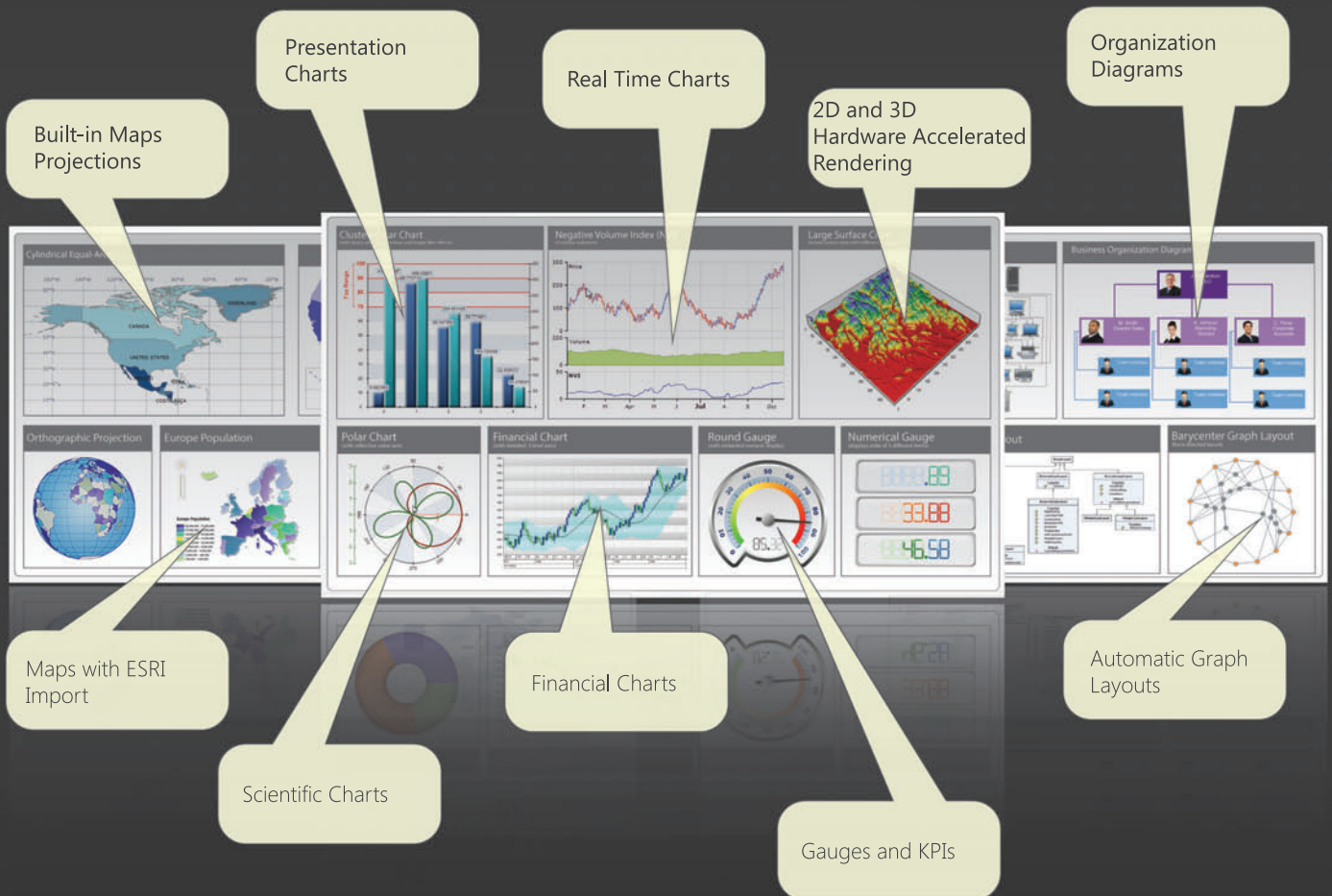
Configuring the PortBridge Agent

Like the PortBridge application, the PortBridge Agent also has a custom configuration section:

```
<portBridgeAgent serviceBusNamespace="countries" serviceBusIssuerName="owner"
serviceBusIssuerSecret="XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
<portMappings>
<port localTcpPort="1433" targetHost="TREDKAR-W530" remoteTcpPort="1433">
<firewallRules>
<rule source="127.0.0.1"/>
<rule sourceRangeBegin="208.0.0.0" sourceRangeEnd="208.255.255.255"/>
...
</firewallRules>
</port>
</portMappings>
```

Nevron Data Visualization

The leading data visualization components for desktop and web development in a single package.



solutions available for

Microsoft
ASP.net

Microsoft
Silverlight

■ ■ **WPF**

■ ■ **WinForms**

■ **SharePoint**

Microsoft
SQL Server

Learn more at www.nevron.com today

www.nevron.com | email@nevron.com | +1 888-201-6088 (Toll free, USA and Canada)

Microsoft, .NET, ASP.NET, SharePoint, SQL Server and Visual Studio are registered trademarks of Microsoft Corporation in the United States and/or other countries. Some Nevron components are only available for certain platforms. For details visit www.nevron.com or send an e-mail to support@nevron.com.



Figure 7 Endpoints and Locations

Solution Component	Input Endpoint	Output Endpoint	Output Endpoint
Countries Web Service	80		Azure Web Site
PortBridge Agent	1433	1433	Azure Virtual Machine
PortBridge	1433	1433	Azure Virtual Machine
SQL Server	1433		Azure Virtual Machine

The PortBridge Agent has a port-mapping section where you map each input endpoint port (localTcpPort) to the Service Bus relay endpoint port (remoteTcpPort). The target host value must match the target host value you created earlier in the PortBridge application configuration. This value is used for connecting to the appropriate Service Bus namespace endpoint. If these two values don't match, the application will not work.

In the firewall rules section, you can explicitly list the IP addresses of machines from which the PortBridge Agent will accept requests. Because my PortBridge Agent will be accepting requests from an Azure Web Site, I need to add the IP ranges of the Azure datacenters to the firewall rules. You can download the Azure datacenter IP ranges from bit.ly/118yDxV.

To deploy PortBridge Agent in Azure, navigate to the Azure portal, create a new Windows VM, copy the PortBridge Agent files and run PortBridgeAgent.exe. As you create the VM, make sure you open the 1433 endpoint for external access so the Azure Web Site can access that port on the PortBridge Agent VM.

Another option for deploying PortBridge Agent is via Apps for Azure, an easy way to deploy Windows Store apps. Apps for Azure is a free app with prepackaged VMs readily available for deployment, and it has a prepackaged PortBridge VM, as shown in **Figure 12**. You can install Apps for Azure from appsforazure.com.

The prepackaged PortBridge Agent VM by default opens the 1433 and 80 endpoint ports for communication. If you want to custom configure the PortBridge Agent, you need to modify the PortBridgeAgent.exe.config file in the C:\ddapplications folder. After configuring the file, you'll have to restart the Dynamic-DeployInitService Windows Service.

Testing and Deploying the Countries Web Service

Once the PortBridge components are installed and running, you'll need to modify the Web service's database connection string to point to the PortBridge Agent VM:

```
<add name="DatabaseConnectionString"
connectionString=
"Server=tejaswiml.cloudapp.net;
Database=cf10292013;
Trusted_Connection=True;" />
```

Note that apart from the host name, all other parameters remain the same. With this configuration change, you'll be pointing to the PortBridge Agent VM instead of the database directly. Next, publish the Countries Web API service to Azure Web Sites and test it by navigating to the following URIs:

1. [http://\[Azure Web site host\]/api/Countries](http://[Azure Web site host]/api/Countries), for retrieving all the countries.
2. [http://\[Azure Web site host\]/api/Countries/\[Country Code\]](http://[Azure Web site host]/api/Countries/[Country Code]), for retrieving a specific country by country code.

If the end-to-end communication is working, both the URIs will return JSON objects for the method invocations. You can now call the Countries Web service from any device and the call will traverse from Azure Web Site to PortBridge Agent, and reach the SQL Server database via the PortBridge application. The data will be retrieved from the database running on your local machine (or datacenter).

Performance Considerations

Because PortBridge is a layer of indirection, it does cause latency when compared to architectures that involve direct database communications or virtual network hybrid connectivity. PortBridge is recommended only in scenarios where virtual network and direct communications aren't feasible. At the time of this writing, Azure Web Sites don't support virtual networking and, therefore, Service Bus is the only option for building hybrid connectivity. During my testing, I saw latencies ranging from 50ms to 300ms for the Countries Web service. For some scenarios, I recommend caching the data in the cloud and only periodically reaching out to the services running in remote datacenters.

Security Considerations

PortBridge relies on the security capabilities of Service Bus Relay. By default, PortBridge uses the connection security to connect to the Service Bus namespace, but it does not employ transport or message encryption capabilities. The PortBridge Agent provides a custom IP address filtering mechanism, but this should not be considered as robust as the built-in security mechanisms of Azure Service Bus. I recommend performing detailed thread modeling before deploying the solution in production.

Scaling Out PortBridge

From the architecture in **Figure 3**, it looks like the PortBridge Agent is a single point of failure. But you can scale out the PortBridge Agent by adding a load-balanced endpoint to the VM and creating multiple instances of the PortBridge Agent VM. The calls coming from the Azure Web Site will then be load balanced across multiple PortBridge Agents. Even though you can scale out PortBridge

Agent, the final destination endpoint (the database) should be designed to handle this scale out. You don't want to scale out the Web and the middle tier but fail on the data tier. Your design should be elastic all the way through.

PortBridge in the Real World

In the past couple of years, based on several customer requests, our team built a messaging hub in the cloud for integrating third-party applications with Azure services. The goal was to create an easy plug-and-play model

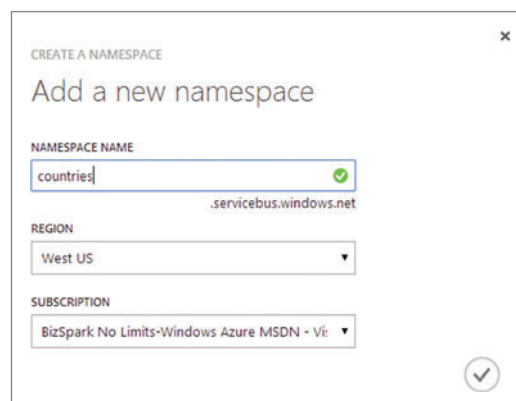


Figure 8 Creating the Service Bus Namespace

Figure 9 Service Bus Namespace Credentials



Figure 10 PortBridge Application Running

for collecting data from disparate data sources and then surface the aggregated data to end users on devices of different sizes. The team decided to use PortBridge as the backbone for creating this plug-and-play model because it provided the necessary abstraction for connecting non-WCF services with applications running in Azure. We did modify the PortBridge code for better diagnostics, performance and caching. Some of the scenarios we successfully implemented were:

1. Creating a unified messaging service that integrated the 19 different data sources of a Fortune 50 company, and providing an on-the-job contextual data mashup to the customer's field employees. Quick access to necessary information in the field was the primary goal of this solution.

NAME	RELAY TYPE	LISTENERS
PortBridge/TREDKAR-W530	NetTcp	1

Figure 11 Service Bus Relay Connection

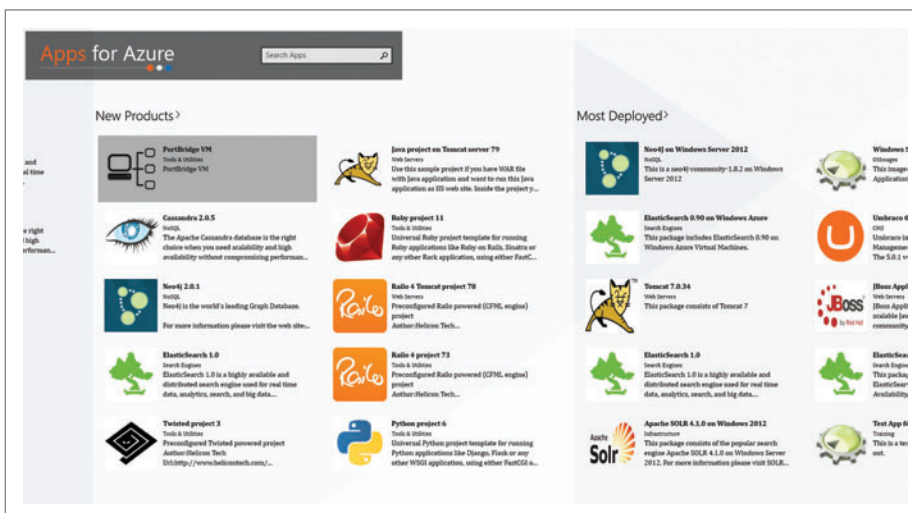


Figure 12 Apps for Azure for Deploying the PortBridge Agent

2. Retrieving data from thousands of application instances used for managing commercial vehicle inventory and storing the data in a central data repository in the cloud. The captured data was used for predictive maintenance and studying buyer behavior. Without Service Bus (and PortBridge), such an application would have taken 10 times the cost and effort.
3. Providing case information and timesheet management to thousands of lawyers on their mobile devices anywhere. Before this application, the lawyers had to travel back to the office after every court case to log their time. With the mobile app, they can log their entries directly from the courthouse.

Wrapping Up

Azure Web Sites and Service Bus complement each other in the building of hybrid Web applications. PortBridge is just an abstraction layer on top of Service Bus Relay for conveniently exposing non-WCF endpoints in the cloud. I have dealt with several scenarios that required building aggregation mashup Web sites where data is retrieved from multiple Web services running at different locations. By using PortBridge, these applications could be designed, tested and built without significant changes to the original application code. Once the initial PortBridge configuration is tested, the connectivity works consistently as long as the Service Bus and Web services are available. Even though PortBridge lets you quickly build hybrid applications, keep in mind it does introduce latency in your service calls. If your application is response-time sensitive, PortBridge may not be the right choice and you should evaluate Azure Virtual Network or migrate on-premises services

to the cloud. After walking through the solution outlined in this article, you should be comfortable externalizing any on-premises endpoint into Azure and then calling it from an Azure Web Site. Source code for PortBridge and the accompanying Countries Web service is available on GitHub at github.com/dynamicdeploy/portbridge. ■

TEJASWI REDKAR is an author and a software developer. He currently works for Microsoft as a director of application platform strategy and communities. His book, "Windows Azure Web Sites: Building Web Apps at a Rapid Pace" (Dynamic Deploy LLC, 2013), is the most comprehensive and best selling on the topic. Redkar is also the creator of appsforazure.com and dynamicdeploy.com, where he experiences first-hand running production apps in the cloud. You can reach out to him at tejaswi_redkar@hotmail.com and follow him on Twitter at twitter.com/tejaswiredkar.

THANKS to the following technical experts for reviewing this article: Microsoft Azure Product Management Team

Teaching from the Cloud

James Chambers

This is an exciting time to be a Web developer. The end-to-end task of creating a project and deploying it to a public-facing endpoint is truly a challenge that was once daunting, perhaps even prohibitive. Yet today stands as one of computing's "solved" challenges.

With very few resources, you can download a free IDE and source control tools. You can start a project and deploy it to a rich infrastructure, maintained for you without needing access to or knowledge of the hardware. You can have your project hosted at no cost to get started.

The building blocks we have to work with are more comprehensive than ever—whether PHP, Java or the Microsoft .NET Framework. We can now focus on the UX instead of project management, networking, storage, deployment procedures or scalability.

FrontClass, the application covered in this article, is something that decades ago would've been weeks of work, if not longer. It would've been difficult to get working in all environments. Deployment would've been a nightmare. And, most likely, I wouldn't have been able to solve scalability in that time frame. Today, I can build the project and have it deployed in hours and ready to scale. The downloadable solution contains the complete, working source code you can deploy without having to modify your own Microsoft Azure account.

I started FrontClass while volunteering at a local junior high school, teaching programming to kids ages 10 to 14. I wanted to share content with students in a way that I could control the pace, but let them go back to previous steps on-demand. There are three functional areas of the application that help a teacher conduct

a class: lesson composition, classroom instruction and student participation. I'll break down each of these areas, but, naturally, there is some overlap.

Project Basics

To build the app, I'll use Visual Studio 2013 Update 2. I'll also use the ASP.NET Web Application when creating the solution and the ASP.NET MVC template. The project spawned from the template will use the popular Bootstrap front-end framework for styling and some UI functionality.

To enable real-time functionality, I'll add the SignalR packages and use tooling now well-known to Visual Studio. I'll also use the requisite jQuery libraries on the client pages. I'll configure the Entity Framework (EF) to use LocalDb by default in the development environment, but when I deploy to Azure Web Sites, I'll be using Azure SQL Database as the data store. I start the whole project, however, in the Azure Portal (portal.azure.com), where I can configure my deployment target.

Create the Azure Web Site At the bottom-left corner of the Portal, click New and select the Website + SQL template, as shown in **Figure 1**. Azure will create a set of linked resources. Select an appropriate name for this group and name the application. I can choose to create a new database server or select an existing one on my account to create the database for my site. It's recommended to have the database and the Web site in the same zone to reduce network traffic times.

In my case, I called the resource group ELearning, the database FrontClass_DB, and the site FrontClass. You'll need to choose a unique site name. Once provisioning is complete, I get the `frontclass.azurewebsites.net` hostname and my deployment target is ready to host my site.

Despite the appearance of magical happenings, there's really not much that should surprise you here. I've created a DNS entry, mapped a host name to an IP address, and made some configuration options like default documents and connection strings. These are essentially the things you'd be doing in IIS behind the scenes to get your site running. So, again, it's not magical, but it's quite convenient. There are also some infrastructure-type tools like source control and deployment scripts available to help get you started.

Create the Solution Next, I head into Visual Studio and select the aforementioned ASP.NET Web Application as a base for my

This article discusses:

- How to architect an educational Microsoft Azure Web Site
- Enabling student authentication and sign-on
- Enabling instructor editing and content development
- Deploying the course content to students

Technologies discussed:

Microsoft Azure Web Sites, ASP.NET MVC 5, Visual Studio, JavaScript, Entity Framework, SignalR

Code download available at:

msdn.microsoft.com/magazine/msdnmag0714

solution. As I choose the type of project to create—using the ASP.NET MVC template—I also get options to create the related resources on Azure. I’ve already created the site and database through the Portal, so I clear the checkbox and proceed with creating the site.

Compose Course Modules

Because of the scope of this project, I’m able to maintain a fairly straightforward data model. I use EF6 to create the models via Code First, which leaves the door open for modifications down the road using data migrations. The end result, shown in **Figure 2**, is a simple set of classes representing the course, module and step structures.

There’s a bit of ceremony to walk through to get all the wiring where I want it. Namely, I want to use a single database, I want to be able to control when migrations are generated and I want my application to automatically execute any outstanding migrations at application startup.

This lets me explicitly control the changes that appear in each migration. It also lets my application update itself without intervention. I will also set up a means to pre-populate the database with the Administrator role and the Instructor role (the first and only administrative user) through the Seed override on the Configuration class.

Specify a Connection String Entity Framework lets me use the same database with different contexts and migrations within a single connection string. By calling the base class constructor and passing in a name, the framework will look first to the application or Web configuration to see if there’s a connection string defined by the same name. If so, it will use it to make the connection.

Migration history for each context is maintained by logging the namespace as part of the update process. This helps my migrations execute independently of each other. To set this up, I added a default constructor to my context class as follows:

```
public FrontClassContext()
: base("DefaultConnection") { }
```

If you look at the context created as part of the project template located in the IdentityModels.cs file, you’ll see there’s a similar default constructor present. It’s slightly modified to reflect the nature of the IdentityDbContext base class, but still uses the DefaultConnection connection string from the Web.Config file.

Enable Migrations I generated the configuration class for the entities created by executing the following command from the Package Manager Console in Visual Studio:

```
Enable-Migrations -ContextTypeName FrontClass.DAL.FrontClassContext
```

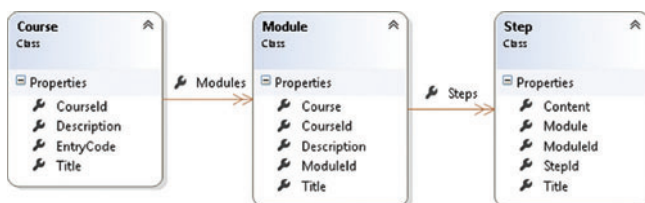


Figure 2 The Basic Application Data Model

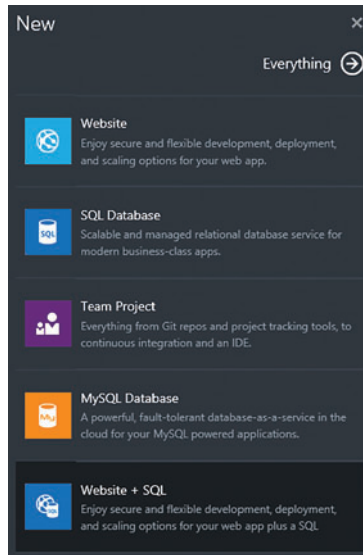


Figure 1 Create a New Azure Web Site with Linked Database

This lights up migrations for my course-related entities. I want to do the same for the ApplicationDbContext, as well. I could run that command again with the class name swapped out. However, when I enable configuration, migrations and seeding in the context where the accounts are stored, I don’t want to overwrite the previously scaffolded configuration class. Instead, I specify an alternate directory for the migrations, as follows, with the MigrationsDirectory parameter passed into the command:

```
Enable-Migrations -ContextTypeName
FrontClass.Models.ApplicationDbContext
-MigrationsDirectory:"Models\AccountMigrations"
```

Add the First Migrations The next step is to simply scaffold the classes the Entity Framework will execute. I use the following command twice, once for each DbContext for which I wish to create a migration:

```
Add-Migration Initial-Model -ConfigurationTypeName
FrontClass.Migrations.Configuration
Add-Migration Initial-Model -ConfigurationTypeName
FrontClass.Models.AccountMigrations.Configuration
```

Again, working with multiple contexts adds a bit of complexity because you have to specify the ConfigurationTypeName parameter. Otherwise, this is a straightforward operation.

Set the Database Initialization Strategy To configure EF to execute my migrations automatically, I need to tell it what strategy to use before performing any database access. Without doing so, I’ll receive exception messages at run time that indicate my model is out of sync with the database. Each change to my classes affects the computed hash of my data model, which is tracked in the database.

In my Global.asax I’ve added the following two lines of code to use the MigrateDatabaseToLatestVersion initializer:

```
Database.SetInitializer<ApplicationDbContext>(
    new MigrateDatabaseToLatestVersion
        <ApplicationDbContext, FrontClass.Models.AccountMigrations.Configuration>());

Database.SetInitializer<FrontClassContext>(
    new MigrateDatabaseToLatestVersion
        <FrontClassContext, FrontClass.Migrations.Configuration>());
```

Note that I’m calling the generic SetInitializer method that accepts the context I want to configure. In my case, my configuration classes are like-named so I’ve specified the fully namespaced class names.

Seed the Tables At first run, I want to be able to deliver an experience that lets someone “just log in” and start using the app. I could create a run-once controller akin to what you’d find in popular blogging applications. Another option is to use the Seed methods in my Configuration classes. The seed method is passed in an instance of the appropriate context, from which I can manipulate the tables.

With ASP.NET Identity 2.0, I also get a rich set of classes that are EF-aware. This lets me conditionally inject the role and user into the database, as I’m doing in **Figure 3**.

I’ve also seeded the module-related tables with some sample data, which you can see in the download that accompanies this article. When the application starts up, I can log in with the instructor@contonso.com username using the password init_2014. At this point, I have my data structures in place, the application is configured with

Figure 3 Seeding the First Role and Administrative User

```
if (!context.Roles.Any(r => r.Name == FrontClass.MvcApplication.
AdministratorRoleName))
{
    var roleStore = new RoleStore<IdentityRole>(context);
    var roleManager = new RoleManager<IdentityRole>(roleStore);
    var identityRole = new IdentityRole { Name = FrontClass.MvcApplication.
AdministratorRoleName };

    roleManager.Create(identityRole);
}

var instructorName = "instructor@contonso.com";
if (!context.Users.Any(u => u.UserName == instructorName))
{
    var userStore = new UserStore<ApplicationUser>(context);
    var userManager = new UserManager<ApplicationUser>(userStore);
    var applicationUser = new ApplicationUser { UserName = instructorName };

    userManager.Create(applicationUser, "init_2014");
    userManager.AddToRole(applicationUser.Id, FrontClass.MvcApplication.
AdministratorRoleName);
}
```

multiple contexts to use the same database and I have an administrative user—the instructor account—that can log in.

Basic Editing Capabilities The last piece required to allow actual course composition is to provide the create, read, update and delete (CRUD) capabilities. I create a new area called administration in the root of my project and use the built-in scaffolding tools to build out the UI. Finally, I create an Admin controller with an index action and a view that provides links to course maintenance, modules and steps.

Instruct the Course

As the instructor, you can choose a course from the same administration view, from which you would then choose a module. Each of the steps are displayed on the subsequent module page, as you can see in **Figure 4**. You can send any step to any student in the virtual classroom with a preview of the content available to the instructor.

The students could be in a conference session, in a classroom with an instructor, or spread around the world. One context in which you could use this application would be a live, online virtual classroom with dozens or even hundreds of students. I want to ensure that regardless of how you may need to adapt the application, you don't lose the ability to scale. So I have to consider how the lesson payload will actually get to the student.

You can create the steps within a course module as plain text or HTML, and each step can be an arbitrary length. SignalR does a great job of handling the mechanics of negotiating transport (with WebSockets, server-sent events, long polling or forever frame) and it abstracts the need to worry about message serialization.

Ultimately, to remain scalable, I need to keep my messages small.

This is to minimize demand on server resources, reduce the costs of serialization, let the client's browser leverage parts of infrastructure such as caching, and use SignalR in the same vein of its design: for signaling.

The best way to achieve this is to simply send a message to the client saying, "New content is available *here*," instead of sending the entire lesson step along. Consider the following serialized message as received by the client in the JSON here:

```
{ "H": "ModuleStepHub", "M": "updateStep",
  "A": [ "\n<h1>Welcome to the course!</h1><p>trimmed for brevity...</p>\n"] ] }
```

That message carries HTML as the argument payload for the client-side updateStep method. That "trimmed for brevity" text is the source of the concern. The content could grow dramatically depending on how an instructor created a step. For now, this is just text. You can imagine how the size of the message would grow if you tried to send pages of content or an image down the pipe. The text or image would be serialized as JSON and delivered to each client using the signaling pipeline resources. Instead, I just want to send the signal, particularly the new content ID the browser should load. That considerably smaller message would look more like this:

```
{ "H": "ModuleStepHub", "M": "notifyStepAvailable", "A": [ 7 ] }
```

Now that I know what I'm trying to accomplish with my messaging, I can build out this functionality in my application.

Create the Hub: I add a Hub folder to my project, then add a SignalR Hub Class (v2) from the Add New Item dialog. A hub is the server-side piece of SignalR you create to publish messages and handle client calls. Going this route, Visual Studio pulls in my dependencies and scaffolds a class with a sample method in my hub. I remove the sample method and create a new one that lets me send a message with the following code:

```
public void MakeStepAvailable(int stepId)
{
    Clients.Others.notifyStepAvailable(stepId);
}
```

The premise here is the course instructor will invoke some kind of action that sends the step along to the students. Using the Others dynamic property on the Clients object, I tell SignalR to send the step ID to all other connected clients. SignalR provides a number of such filtering options so I can update a specific user, a group of

Course Instruction for Learning Math

Current Module: Math Basics

Description:

Instructions: Select a step below to update all students with the content for that step.

Addition

Share

Subtraction

Share

Multiplication

Share

Division

Share

Order of Operations

Share

Student Chalkboard Preview

Module Title Math Basics

Title Addition

Welcome to the course!

So, you've decided to learn math. Good on ya! You'll find that it's as easy as 1-2-3.

This course assumes that you have number recognition down pat, and understand the mechanics of counting.

Let's start with one of the most basic parts of math, addition.

Addition

Also known as 'adding', addition refers to computing the sum of two or more numbers.

Figure 4 Conduct the Class

Figure 5 JavaScript to Start up the Proxy and Handle Click Events

```
<script src="../../Scripts/jquery.signalR-2.0.3.js"></script>
<script src="../../SignalR/Hubs"></script>
<script>
    $(function() {
        var hub = $.connection.moduleStepHub;

        $.connection.hub.start().done(function() {
            $(".share-step").click(function() {
                var stepId = $(this).attr("data-id");
                hub.server.makeStepAvailable(stepId);
            });
        });
    });
</script>
```

users, or some other slice of connected clients. The hub also lets instructors start a new module. Students can get the list of steps made available by the instructor.

Map SignalR Hubs To expose the functionality I create in my hub, I need to poke SignalR as my application is starting. A call to MapSignalR sets up a default endpoint that serves up a JavaScript proxy. When included in the client, this invokes server-side methods from the client and vice versa. The root folder of my project includes an OWIN Startup class where I make the call to do the wiring in the Configuration method. This is how that looks after my changes:

```
public void Configuration(IAppBuilder app)
{
    ConfigureAuth(app);
    app.MapSignalR();
}
```

It's important to call ConfigureAuth here first. The OWIN pipeline pushes messages through middleware in the order of which the middleware is registered. In our case, we want authentication and authorization to occur before calls reach our Hub.

Enable Sharing To let the instructor share content, I add another controller to the application called InstructController. This has only an Index action. I've decorated the class with the Authorize attribute, so only Administrators can control the classroom. The index action accepts a module ID as a parameter, the route for which is configured in my App_Startup\RouteConfig.cs class. The method looks up the module from the database and returns the record to the view with the Steps collection included in the query.

In the corresponding view (located at Views\Instruct\Index.cshtml), I simply generate a Share button for each step. I use a data-id attribute to store the ID of the step. I include the SignalR library and the hub proxy exposed in the previous step, then write a small amount of JavaScript to start up the proxy and handle click events from the instructor, as shown in **Figure 5**.

Finally, I modify my administrative index page to include the list of courses and modules from which the instructor can choose. The final interface, as displayed in **Figure 6**, lets the instructor jump right into a module and start teaching.

Using the module view from **Figure 4**, the instructor can then send content to the class.

Class Participation

As a student, the story is much simpler. Just show up! Each student will need to sign up to access the site, and receive instructions for the active course module they need to enter

Manage Your Courses

Create new or modify existing [courses](#).

Manage your [modules](#) for any course.

Create [content](#) for your modules.

Begin Instruction

Click on any module below to start a class and begin instruction.

Learning Math

Math Basics

Fun with Hexadecimal

Statistics and You

Calculus for Fun and Profit

Figure 6 The Administrative Interface for FrontClass

the passcode. The registration process is taken care of by the default Account controller in my project. I'll need to create the page that hosts the lesson content and write the code needed for students to participate in the class. For students to join the classroom when a class is active, they must have first enrolled for the course.

Build the Classroom Controller Right-clicking on the Controllers folder in the root of the project lets me add a new controller and name it ClassroomController. The two primary methods I add to this class are Index, where the main content will be served, and GetModuleStep, which returns the requested step after meeting a number of preconditions. A third action handles cases where there's no active module for the class. To access module materials, a student must be logged in to the site and enrolled in the active module's course.

Allow Course Enrollment The ClassroomController is decorated with an EnrollmentVerificationFilter I created to ensure students can only access courses for which they have the entry code set by the instructor. If the user hasn't yet enrolled or an instructor hasn't started a module, then students are redirected to the Index action on the EnrollmentController, as you can see in **Figure 7**. Students can enroll in a course with the appropriate entry code. Upon successful enrollment, I add a claim to the user account using the relevant ASP.NET Identity components.

I add a claim through the UserManager class, to which I gain access by getting an instance of my ApplicationDbContext and using that to build a UserStore, which is then passed into my target class:

```
var context = new ApplicationDbContext();
var userStore = new UserStore<ApplicationUser>(context);
var userManager = new UserManager<ApplicationUser>(userStore);
```

Once I have the necessary component in place, I get the user's ID through the principal representing his identity and build a claim object. Then I add that to the user's account:

```
var userId = User.Identity.GetUserId();
var courseClaim = new Claim(MvcApplication.CourseRegistrationClaimUrn,
    enroll.CourseId.ToString(CultureInfo.InvariantCulture));
userManager.AddClaim(userId, courseClaim);
```

Course Lobby and Enrollment

Ready to start? Make sure you're enrolled in "Learning Math" below and then [enter the Classroom »](#)

Course

Enrolment Status

Learning Math

[+ Enrol](#)

Figure 7 The Course Lobby and Enrollment Page

With the claim in place on the user's account, a student may now access the classroom.

Examine the Classroom View As you can see in **Figure 8**, the classroom is divided into two parts. The top of the page gives an overview of the current module and provides controls to jump to any available step. The bottom section is called the chalkboard, and displays the content shared by the instructor.

This page really only has a couple of DIV elements as content containers. The rest is wired up via JavaScript and through content fetched based on messages coming from the ClassroomHub or the student's selection of a previous step. As the instructor selects new steps, the ClassroomHub is notified and sends signals to all who are in the classroom. This in turn fetches data from the ClassroomController, thus enabling caching.

Deploy the Project

The instructor can create and manage courses, modules and steps. A student can register for the site and enroll himself in courses. Courses are protected through enrollment claims, and the classroom's virtual chalkboard is updated with content as directed by the instructor. Time to ship!

From the Build menu, I select the option to publish my project. The Publish Web dialog is displayed, and I choose Azure Web Sites as the publish target. Visual Studio prompts me for my credentials if I haven't already signed in. Then I can choose the existing site I created at the start of this project. My publishing profile is then downloaded, complete with the pre-configured database connection strings and all credentials required to send my site to Azure.

At the beginning of this project, I first prepped my site in the Azure Portal. Because this project uses the DefaultConnection connection string, already present in my publishing profile, I don't need to make any changes when I go to production. The production connection string automatically points my application at the previously configured Azure SQL Database.

Take Advantage of Azure Web Sites

Publishing a site to a pre-configured endpoint in Azure Web Sites was a welcome surprise. It was actually easy. After years of surviving with much more complicated deployment procedures, this was surely a treat. This alone is not the end game for Azure Web Sites. There are many more tools available:

- **Add a Custom Domain:** If you can register a domain and set up a couple of DNS records, it only takes minutes to add a unique domain of your choosing to your site. Read more about this at bit.ly/1sV8R1y.
- **Secure the Application:** SNI-based certificates are affordable, and you can add them through the dashboard of your Azure Web Site. Read more about this at bit.ly/1mYXndJ.
- **Scale Up the Site:** If you find your project is growing, you have the option to scale up (more powerful hardware) or out (more instances). Don't forget to also configure Azure

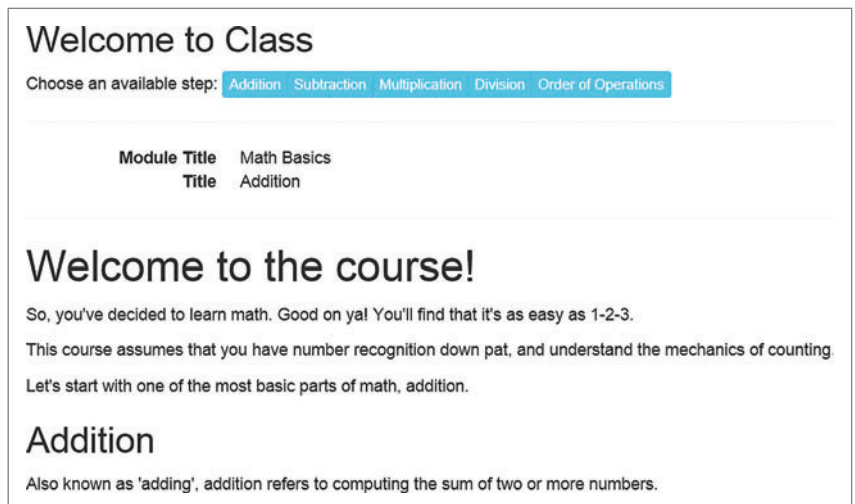


Figure 8 The Virtual Chalkboard

Service Bus to handle your SignalR traffic. Read more about SignalR at bit.ly/1o6B7AC.

- **Add Continuous Deployment:** For even the smallest projects, I forego direct deployment and use a Git-based source control server. Azure then creates, stages and automatically deploys my site on check-in. Read more about this at bit.ly/1o6BACT.
- **Add Monitoring and Alerts:** It's always best to know when something's not going quite right in your application. You can watch for certain types of problems and get notifications as they happen, all configured from your site's dashboard.

Wrapping Up

In this article, I used ASP.NET Identity 2.0, which is now part of the default ASP.NET MVC template in Visual Studio 2013. To learn more about the enhancements and changes, please check out the post on the .NET Web Development and Tools Blog at bit.ly/PXgQ2d. I also used SignalR to enable real-time communication between the client and the server. This automatically generates JavaScript proxies and invokes code on the server from the client and vice versa. For more on creating your data model using EF with Code First in an ASP.NET MVC 5 application, visit the tutorial at bit.ly/1pivbmE.

Rich experiences are a lot easier to deliver when you don't have to take care of the underlying infrastructure or the complexities of deployment. The FrontClass application leverages authentication, authorization, real-time messaging and a persistent data store. As a developer, I get many of these building blocks thanks to the efforts of others. I don't have to worry about the other moving parts as I move my project into production. ■

JAMES CHAMBERS is a Microsoft MVP in ASP.NET/IIS and frequent speaker at conferences and user groups across Canada. He's the author of "Windows Azure Web Sites" (Wrox, 2013), available via eBook (bit.ly/wawsbook), and presented on the Microsoft Virtual Academy (bit.ly/wawsmva). He blogs at jameschambers.com and you can reach him on Twitter at twitter.com/CanadianJames.

THANKS to the following technical expert for reviewing this article:
Chad McCallum (independent consultant)



For 21 years, Visual Studio Live! has been the trusted source of independent, practical training on the Microsoft Platform for developers all over the world.

vslive.com

THREE MORE EVENTS REMAIN IN 2014: WHICH ONE WILL YOU ATTEND?

REDMOND 2014

August 18 – 22

Microsoft Headquarters

Redmond, WA

(see pages 62 & 63 for more details)



WASHINGTON, D.C.

October 6 – 9, 2014

Washington Marriott at Metro Center

(see pages 64 & 65 for more details)



ORLANDO

November 17-21, 2014

Loews Royal Pacific Resort

at Universal Orlando

Part of Live! 360

(see pages 24 & 25 for more details)



CONNECT WITH VISUAL STUDIO LIVE!



twitter.com/vslive – @VSLive



facebook.com – Search “VSLive”



linkedin.com – Join the
“Visual Studio Live” group!

TURN THE PAGE FOR
MORE EVENT DETAILS



Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

YOUR GUIDE TO THE .NET DEVELOPMENT UNIVERSE

REDMOND 2014

August 18 – 22 | Microsoft Headquarters
Redmond, WA



Explore HOT TOPICS like:

- UX Design
- Angular
- Entity Framework
- Xamarin
- And Many More!

SET YOUR COURSE FOR 127.0.0.1!

From August 18 – 22, 2014, developers, software architects, engineers and designers will land in Redmond, WA at the idyllic Microsoft headquarters for 5 days of cutting-edge education on the Microsoft Platform.

Register by July 16 and Save \$300!

Use promo code REDJUL2



Scan the QR code to
register or for more
event details.

vslive.com/redmond

EVENT SPONSOR
Microsoft

PLATINUM SPONSOR
 **esri**

SPONSOR
 **LogiGear**
Software Testing

SUPPORTED BY
 **Visual Studio**

 **msdn**
magazine

Visual Studio
MAGAZINE

PRODUCED BY
 **1105 MEDIA**

Redmond AGENDA AT-A-GLANCE

Visual Studio / .NET Framework	Windows Client	Cloud Computing	Windows Phone	Cross-Platform Mobile Development	ASP.NET	JavaScript / HTML5 Client	SharePoint	SQL Server
--------------------------------	----------------	-----------------	---------------	-----------------------------------	---------	---------------------------	------------	------------

START TIME	END TIME	Visual Studio Live! Pre-Conference Workshops: Monday, August 18, 2014 (Separate entry fee required)		
7:00 AM	8:00 AM	Pre-Conference Workshop Registration - Coffee and Morning Pastries		
8:00 AM	12:00 PM	MW01 - Workshop: Modern UX Design - <i>Billy Hollis</i>	MW02 - Workshop: Data-Centric Single Page Applications with Durandal, Knockout, Breeze, and Web API - <i>Brian Noyes</i>	MW03 - Workshop: SQL Server for Developers - <i>Andrew Brust & Leonard Label</i>
12:00 PM	2:30 PM	Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center		
2:30 PM	6:00 PM	MW01 - Workshop: Modern UX Design - <i>Billy Hollis</i>	MW02 - Workshop: Data-Centric Single Page Applications with Durandal, Knockout, Breeze, and Web API - <i>Brian Noyes</i>	MW03 - Workshop: SQL Server for Developers - <i>Andrew Brust & Leonard Label</i>
7:00 PM	9:00 PM	Dine-A-Round Dinner		

START TIME	END TIME	Visual Studio Live! Day 1: Tuesday, August 19, 2014				
7:30 AM	8:30 AM	Registration - Coffee and Morning Pastries				
8:30 AM	9:30 AM	Keynote: Microsoft <div>UNDER CONSTRUCTION</div>				
9:45 AM	11:00 AM	T01 - Microsoft Session: Details to Come <div>UNDER CONSTRUCTION</div>	T02 - Creating Data-Driven Mobile Web Apps with ASP.NET MVC and jQuery Mobile - <i>Rachel Appel</i>	T03 - HTML5 for Better Web Sites - <i>Robert Boedigheimer</i>	T04 - Introduction to Windows Azure - <i>Vishwas Lele</i>	T05 - What's New in the Visual Studio 2013 IDE
11:15 AM	12:30 PM	T06 - What's New in WinRT Development - <i>Rockford Lhotka</i>	T07 - Microsoft Session: Details to Come <div>UNDER CONSTRUCTION</div>	T08 - Great User Experiences with CSS 3 - <i>Robert Boedigheimer</i>	T09 - Windows Azure Cloud Services - <i>Vishwas Lele</i>	T10 - What's New for Web Developers in Visual Studio this Year? - <i>Mads Kristensen</i>
12:30 PM	2:30 PM	Lunch - Visit Exhibitors				
2:30 PM	3:45 PM	T11 - Interaction Design Principles and Patterns - <i>Billy Hollis</i>	T12 - Getting Started with Xamarin - <i>Walt Ritscher</i>	T13 - Building Real Time Applications with ASP.NET SignalR - <i>Rachel Appel</i>	T14 - Microsoft Session: Details to Come <div>UNDER CONSTRUCTION</div>	T15 - Why Browser Link Changes Things, and How You Can Write Extensions? - <i>Mads Kristensen</i>
3:45 PM	4:15 PM	Sponsored Break - Visit Exhibitors				
4:15 PM	5:30 PM	T16 - Applying UX Design in XAML - <i>Billy Hollis</i>	T17 - Building Multi-Platform Mobile Apps with Push Notifications - <i>Nick Landry</i>	T18 - JavaScript for the C# Developer - <i>Philip Japikse</i>	T19 - Windows Azure SQL Database – SQL Server in the Cloud - <i>Leonard Lobel</i>	T20 - Katana, OWIN, and Other Awesome Codenames: What's Coming? - <i>Howard Dierking</i>
5:30 PM	7:00 PM	Microsoft Ask the Experts & Exhibitor Reception – Attend Exhibitor Demos				

START TIME	END TIME	Visual Studio Live! Day 2: Wednesday, August 20, 2014					
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries					
8:00 AM	9:00 AM	Keynote: Microsoft UNDER CONSTRUCTION					
9:15 AM	10:30 AM	W01 - Getting Started with Windows Phone Development - <i>Nick Landry</i>	W02 - Learning Entity Framework 6 - <i>Leonard Label</i>	W03 - Build Data-Centric HTML5 Single Page Applications with Breeze - <i>Brian Noyes</i>	W04 - What's New for XAML Windows Store Apps - <i>Ben Dewey</i>	W05 - Upgrading Your Existing ASP.NET Apps - <i>Pranav Rastogi</i>	
10:45 AM	12:00 PM	W06 - Build Your First Mobile App in 1 Hour with Microsoft App Studio - <i>Nick Landry</i>	W07 - Programming the T-SQL Enhancements in SQL Server 2012 - <i>Leonard Label</i>	W08 - Knocking it Out of the Park, with Knockout.JS - <i>Miguel Castro</i>	W09 - From the Internet of Things to Intelligent Systems: A Developer's Primer - <i>Rick Garibay</i>	W10 - Creating Map Centric Applications for Windows, WinRT and Windows Phone - <i>Ben Ramseth</i>	
12:00 PM	1:30 PM	Birds-of-a-Feather Lunch - Visit Exhibitors					
1:30 PM	2:45 PM	W11 - What's New for HTML/WinJS Windows Store Apps - <i>Ben Dewey</i>	W12 - SQL Server 2014: Features Drill-down - <i>Scott Klein</i>	W13 - JavaScript: Turtles, All the Way Down - <i>Ted Neward</i>	W14 - Zero to Connected with Windows Azure Mobile Services - <i>Brian Noyes</i>	W15 - Microsoft Session: Details to Come UNDER CONSTRUCTION	
2:45 PM	3:15 PM	Sponsored Break - Exhibitor Raffle @ 2:55 pm (Must be present to win)					
3:15 PM	4:30 PM	W16 - Windows 8 HTML/JS Apps for the ASP.NET Developer - <i>Adam Tuliper</i>	W17 - SQL Server 2014 In-memory OLTP - Deep Dive - <i>Scott Klein</i>	W18 - AngularJS JumpStart - <i>Brian Noyes</i>	W19 - Leveraging Azure Web Sites - <i>Rockford Lhotka</i>	W20 - ALM with Visual Studio 2013 and Team Foundation Server 2013 - <i>Brian Randall</i>	
8:00 PM	10:00 PM	Lucky Strike Evening Out Party					

START TIME	END TIME	Visual Studio Live! Day 3: Thursday, August 21, 2014					
7:30 AM	8:00 AM	Registration - Coffee and Morning Pastries					
8:00 AM	9:15 AM	TH01 - Developing Awesome 3D Applications with Unity and C#/JavaScript - <i>Adam Tulper</i>	TH02 - AWS for the SQL Server Professional - <i>Lynn Langit</i>	TH03 - Sexy Extensibility Patterns - <i>Miguel Castro</i>	TH04 - Building APIs with NodeJS on Microsoft Azure Websites - <i>Rick Garibay</i>	TH05 - Leveraging Visual Studio Online - <i>Brian Randall</i>	
9:30 AM	10:45 AM	TH06 - What's New in WPF 4.5 - <i>Walt Ritscher</i>	TH07 - Real-world Predictive Analytics with PowerBI and Predixion Software - <i>Lynn Langit</i>	TH08 - What's New in MVC 5 - <i>Miguel Castro</i>	TH09 - Solving Security and Compliance Challenges with Hybrid Clouds - <i>Eric D. Boyd</i>	TH10 - Essential C# 6.0 - <i>Mark Michaelis</i>	
11:00 AM	12:15 PM	TH11 - Implementing M-V-VM (Model-View-View Model) for WPF - <i>Philip Japikse</i>	TH12 - Excel, Power BI and You: An Analytics Superhub - <i>Andrew Brust</i>	TH13 - What's New in Web API 2 - <i>Miguel Castro</i>	TH14 - Building Mobile Applications with SharePoint 2013 - <i>Darrin Bishop</i>	TH15 - Performance and Diagnostics Hub in Visual Studio 2013 - <i>Brian Peek</i>	
12:15 PM	2:15 PM	Lunch @ The Mixer - Visit the Microsoft Company Store & Visitor Center					
2:15 PM	3:30 PM	TH16 - Build Maintainable Windows Store Apps with MVVM and Prism - <i>Brian Noyes</i>	TH17 - Big Data 101 with HDInsight - <i>Andrew Brust</i>	TH18 - Finding and Consuming Public Data APIs - <i>G. Andrew Duthie</i>	TH19 - Building Apps for SharePoint - <i>Mark Michaelis</i>	TH20 - Git for the Microsoft Developer - <i>Eric D. Boyd</i>	
3:45 PM	5:00 PM	TH21 - Make Your App Alive with Tiles and Notifications - <i>Ben Dewey</i>	TH22 - NoSQL for the SQL Guy - <i>Ted Neward</i>	TH23 - Provide Value to Customers and Enhance Site Stickiness By Creating an API - <i>G. Andrew Duthie</i>	TH24 - Data as Information: Understanding Your SharePoint 2013 Business Intelligence Options - <i>Darrin Bishop</i>	TH25 - Writing Asynchronous Code Using .NET 4.5 and C# 5.0 - <i>Brian Peek</i>	

START TIME	END TIME	Visual Studio Live! Post-Conference Workshops: Friday, August 22, 2014 (Separate entry fee required)	
7:30 AM	8:00 AM	Post-Conference Workshop Registration - Coffee and Morning Pastries	
8:00 AM	12:00 PM	FW01 - Workshop: Service Orientation Technologies: Designing, Developing, & Implementing WCF and the Web API - <i>Miguel Castro</i>	FW02 - Workshop: A Day of Windows Azure - <i>Eric D. Boyd</i>
12:00 PM	1:00 PM	Lunch	
1:00 PM	5:00 PM	FW01 - Workshop: Service Orientation Technologies: Designing, Developing, & Implementing WCF and the Web API - <i>Miguel Castro</i>	FW02 - Workshop: A Day of Windows Azure - <i>Eric D. Boyd</i>

Speakers and sessions subject to change

Visual Studio **LIVE!**

EXPERT SOLUTIONS FOR .NET DEVELOPERS

WASHINGTON, D.C.

October 6 – 9, 2014

Washington Marriott at Metro Center



TO BOLDLY
CODE

WHERE NO VISUAL STUDIO LIVE!
HAS CODED BEFORE

**YOUR GUIDE TO THE
.NET DEVELOPMENT
UNIVERSE**



**Register by
August 13
and Save \$300!**

Use promo code DCJUL2



Scan the QR code to
register or for more
event details.

Use code DCJUL2

vslive.com/dc

**THAT'S RIGHT, WE'RE TRANSPORTING
Visual Studio Live! to our nation's capital for the
first time in 21 years. From Oct 6 – 9, 2014,
developers, software architects, engineers and
designers will gather for 4 days of cutting-edge
education on the Microsoft Platform.**

Explore HOT TOPICS like:

- Visual Studio 2013 IDE
- MVC 5
- Breeze
- C# 6.0
- And Many More!

SUPPORTED BY

Microsoft



Visual Studio

msdn
magazine

Visual Studio
MAGAZINE

PRODUCED BY

1105 MEDIA

Washington D.C. AGENDA AT-A-GLANCE

Visual Studio / .NET	Windows Client	Cloud Computing	Windows Phone	Cross-Platform Mobile Development	ASP.NET	JavaScript / HTML5 Client	SharePoint / Office	SQL Server
START TIME	END TIME	Visual Studio Live! Pre-Conference Workshops: Monday, October 6, 2014 (Separate entry fee required)						
7:30 AM	9:00 AM	Pre-Conference Workshop Registration						
9:00 AM	6:00 PM	MW01 - Workshop: Deep Dive Into Visual Studio 2013, TFS, and Visual Studio Online - <i>Brian Randell</i>		MW02 - Workshop: SQL Server for Developers - <i>Andrew Brust & Leonard Lobel</i>		MW03 - Workshop: Data-Centric Single Page Applications with Angular, Breeze, and Web API - <i>Brian Noyes</i>		
6:00 PM	9:00 PM	Dine-A-Round Dinner						
START TIME	END TIME	Visual Studio Live! Day 1: Tuesday, October 7, 2014						
7:00 AM	8:00 AM	Registration						
8:00 AM	9:00 AM	Keynote: To Be Announced						
9:15 AM	10:30 AM	T01 - Great User Experiences with CSS 3 - <i>Robert Boedigheimer</i>	T02 - What's New in MVC 5 - <i>Miguel Castro</i>		T03 - New IDE and Editor Features in Visual Studio 2013 - <i>Deborah Kurata</i>		T04 - Creating Universal Windows Apps for Business - <i>Rockford Lhotka</i>	
10:45 AM	12:00 PM	T05 - Using jQuery to Replace the Ajax Control Toolkit - <i>Robert Boedigheimer</i>	T06 - ASP.NET Reloaded: Web Forms vs. MVC vs. Web API - <i>Dino Esposito</i>		T07 - Introduction to In Memory OLTP Using Hekaton in SQL Server 2014 - <i>Kevin Goff</i>		T08 - WPF Data Binding in Depth - <i>Brian Noyes</i>	
12:00 PM	1:30 PM	Lunch - Visit Exhibitors						
1:30 PM	2:45 PM	T09 - Build an Angular and Bootstrap Web Application in Visual Studio from the Ground Up - <i>Deborah Kurata</i>	T10 - What's New in Web API 2 - <i>Miguel Castro</i>		T11 - Making the Most of the Visual Studio Online - <i>Brian Randell</i>		T12 - Moving Web Apps to the Cloud - <i>Eric D. Boyd</i>	
3:00 PM	4:15 PM	T13 - JavaScript for the C# (and Java) Developer - <i>Philip Japikse</i>	T14 - Never Mind the Mobile Web; Here's the Device Web - <i>Dino Esposito</i>		T15 - Cross-platform Dev (iOS, Android and Java) with TFS and Team Explorer Everywhere - <i>Brian Randell</i>		T16 - Solving Security and Compliance Challenges with Hybrid Clouds - <i>Eric D. Boyd</i>	
4:15 PM	5:45 PM	Exhibitor Welcome Reception						
START TIME	END TIME	Visual Studio Live! Day 2: Wednesday, October 8, 2014						
7:00 AM	8:00 AM	Registration						
8:00 AM	9:00 AM	Keynote: To Be Announced						
9:00 AM	9:30 AM	Networking Break						
9:30 AM	10:45 AM	W01 - Writing Next Generation JavaScript with TypeScript - <i>Rachel Appel</i>	W02 - A Survey of Two Popular Visual Studio Tools - Web Essentials and NuGet - <i>John Petersen</i>		W03 - Getting Started with Xamarin - <i>Walt Ritscher</i>		W04 - SQL for Application Developers - Attendees Choose - <i>Kevin Goff</i>	
11:00 AM	12:15 PM	W05 - Building Single-Page Web Applications Using Kendo UI and the MVVM Pattern - <i>Ben Hoelting</i>	W06 - ASP.NET MVC - Rudiments of Routing - <i>Walt Ritscher</i>		W07 - Getting Started with Windows Phone Development - <i>Nick Landry</i>		W08 - Database Development with SQL Server Data Tools - <i>Leonard Lobel</i>	
12:15 PM	1:30 PM	Birds-of-a-Feather Lunch - Visit Exhibitors						
1:30 PM	2:45 PM	W09 - Introduction to AngularJS - <i>John Petersen</i>	W10 - Slice Development Time with ASP.NET MVC, Visual Studio, and Razor - <i>Philip Japikse</i>		W11 - The Great Mobile Debate: Native vs. Hybrid App Development - <i>Nick Landry</i>		W12 - Learning Entity Framework 6 - <i>Leonard Lobel</i>	
3:00 PM	4:15 PM	W13 - Creating Angular Applications Using Visual Studio LightSwitch - <i>Michael Washington</i>	W14 - Build Data Driven Web Sites with WebMatrix 3 and ASP.NET Web Pages - <i>Rachel Appel</i>		W15 - Busy .NET Developer's Guide to iOS - <i>Ted Neward</i>		W16 - Team Foundation Server for Scrum Teams - <i>Richard Hundhausen</i>	
4:30 PM	5:45 PM	W17 - To Be Announced	W18 - Build Real Time Websites and Apps with SignalR - <i>Rachel Appel</i>		W19 - Busy .NET Developer's Guide to Android - <i>Ted Neward</i>		W20 - Team Foundation Server: Must-Have Tools and Widgets - <i>Richard Hundhausen</i>	
START TIME	END TIME	Visual Studio Live! Day 3: Thursday, October 9, 2014						
7:30 AM	8:00 AM	Registration						
8:00 AM	9:15 AM	TH01 - Excel, Power BI and You: An Analytics Superhub - <i>Andrew Brust</i>	TH02 - What's New in WPF 4.5 - <i>Walt Ritscher</i>		TH03 - Visual Studio 2013, Xamarin and Windows Azure Mobile Services: A Match Made in Heaven - <i>Rick Garibay</i>		TH04 - What's New in the .NET 4.5.1 BCL - <i>Jason Bock</i>	
9:30 AM	10:45 AM	TH05 - Big Data 101 with HDInsight - <i>Andrew Brust</i>	TH06 - WPF for the Real World - <i>Brian Lagunas</i>		TH07 - Building Your First Windows Phone 8.1 Application - <i>Brian Peek</i>		TH08 - Essential C# 6.0 - <i>Mark Michaelis</i>	
11:00 AM	12:15 PM	TH09 - Cloud or Not, 10 Reasons Why You Must Know "Web Sites" - <i>Vishwas Lele</i>	TH10 - Deploying WinRT Apps Without the Store - <i>Rockford Lhotka</i>		TH11 - Building Games for Windows and Windows Phone - <i>Brian Peek</i>		TH12 - To Be Announced	
12:15 PM	1:15 PM	Lunch						
1:15 PM	2:30 PM	TH13 - Loosely Coupled Applications with Service Bus and Document-centric Data Stores - <i>Vishwas Lele</i>	TH14 - Creating Cross Platform Games with Unity - <i>Brian Lagunas</i>		TH15 - Visual Studio Cloud Business Apps - <i>Michael Washington</i>		TH16 - Asynchronous Debugging in .NET - <i>Jason Bock</i>	
2:45 PM	4:00 PM	TH17 - Not Your Father's BizTalk Server: Building Modern Hybrid Apps with Windows Azure BizTalk Services - <i>Rick Garibay</i>	TH18 - Use Your .NET Code in WinRT with Brokered Assemblies - <i>Rockford Lhotka</i>		TH19 - Best Practices: Building Apps for Office Using HTML/JavaScript - <i>Mark Michaelis</i>		TH20 - Adventures in Unit Testing: TDD vs. TED - <i>Ben Hoelting</i>	
4:15 PM	5:15 PM	Conference Wrap-Up - <i>Andrew Brust (Moderator), Jason Bock, Ben Hoelting, Rockford Lhotka, & Brian Peek</i>						

Sessions and speakers subject to change



Distorting the MNIST Image Data Set

The mixed National Institute of Standards and Technology (MNIST) data set is a collection of 70,000 small images of handwritten digits. The data was created to act as a benchmark for image recognition algorithms. Even though MNIST images are small (28 x 28 pixels), and there are only 10 possible digits (zero through nine) to recognize, and there are 60,000 training images for creating an image recognition model (with 10,000 images held out to test the accuracy of a model), experience has shown that recognizing the MNIST images is a difficult problem.

A clever way to
programmatically generate
more training data is to distort
each original image.

One way to deal with difficult pattern classification problems, including image recognition, is to use more training data. And a clever way to programmatically generate more training data is to distort each original image. Take a look at the demo program in **Figure 1**. The figure shows the digit 4 in original MNIST form on the left, and the digit after distortion using elastic deformation on the right. The parameters in the upper-right corner of the demo application indicate the distortion depends on values for a displacement randomization seed, a kernel size and standard deviation, and an intensity.

It's unlikely you'll need to distort images in most work environments, but you might find the information in this article useful for three reasons. First, understanding exactly how image distortion works by seeing actual code will help you understand many image-recognition articles. Second, several of the programming techniques used in image distortion can be useful in other, more common programming scenarios. And third, you might just find image distortion an interesting topic for its own sake.

This article assumes you have advanced-level programming skills but doesn't assume you know anything about image distortion. The demo program is coded in C# and makes extensive use

of the Microsoft .NET Framework so refactoring to a non-.NET language would be challenging. The demo has most normal error checking removed to keep the size of the code small and the main ideas clear. Because the demo is a Windows Form application created by Visual Studio, much of the code is related to the UI and is spread over several files. However, I've refactored the demo code to a single C# source code file, which is available at msdn.microsoft.com/magazine/msdnmag0714. You can find the MNIST data in several places on the Internet. The main repository is at yann.lecun.com/exdb/mnist.

Overall Program Structure

To create the demo program I launched Visual Studio and created a Windows Form application named *MnistDistort*. The UI has eight TextBox controls for the paths to the unzipped MNIST data files (one pixel file, one label file); the indices of the currently displayed and next image; and a seed value, kernel size, kernel standard deviation and intensity value for the distortion process. A DropDown control holds values to magnify the image view. There are three Button controls to load the 60,000 MNIST images and labels into memory, display an image, and distort the displayed image. There are two PictureBox controls to display regular and distorted images. Finally, a ListBox control is used to display progress and logging messages.

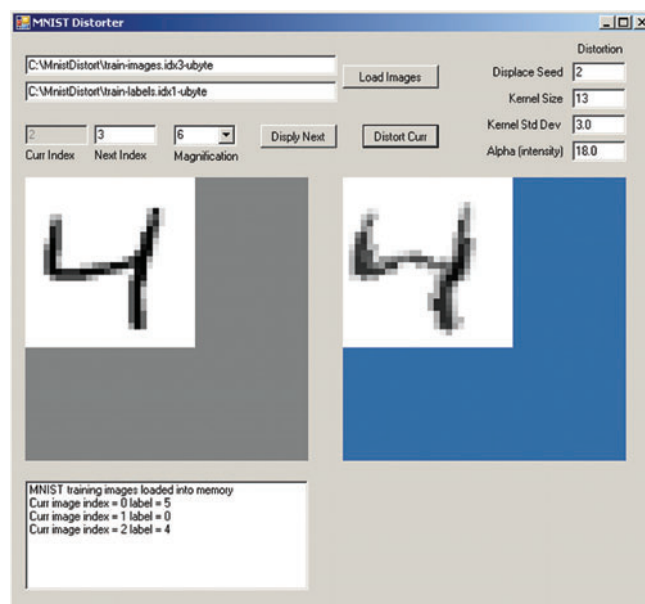


Figure 1 A Distorted MNIST Image

Code download available at msdn.microsoft.com/magazine/msdnmag0714.

Figure 2 Image Distortion Calling Code

```
private void button3_Click(object sender, EventArgs e)
{
    int currIndex = int.Parse(textBox3.Text);
    int mag = int.Parse(comboBox1.SelectedItem.ToString());
    int kDim = int.Parse(textBox5.Text); // Kernel dimension
    double kStdDev = double.Parse(textBox6.Text); // Kernel std dev
    double intensity = double.Parse(textBox7.Text);
    int rndSeed = int.Parse(textBox8.Text); // Randomization

    DigitImage currImage = trainImages[currIndex];
    DigitImage distorted = Distort(currImage, kDim, kStdDev,
        intensity, rndSeed);
    Bitmap bitMapDist = MakeBitmap(distorted, mag);
    pictureBox2.Image = bitMapDist;
}
```

At the top of the source code I removed references to unneeded namespaces and added a reference to the System.IO namespace to be able to read the MNIST data files.

I added a class-scope array named `trainImages`, which holds references to program-defined `DigitImage` objects, and variables to hold the locations of the two MNIST data files:

```
DigitImage[] trainImages = null;
string pixelFile = @"C:\MnistDistort\train-images.idx3-ubyte"; // Edit
string labelFile = @"C:\MnistDistort\train-labels.idx1-ubyte"; // Edit
```

In the Form constructor I added these six lines of code:

```
textBox1.Text = pixelFile;
textBox2.Text = labelFile;
comboBox1.SelectedItem = "6"; // Magnification
textBox3.Text = "NA"; // Curr index
textBox4.Text = "0"; // Next index
this.ActiveControl = button1;
```

The `Button1` click event handler loads the 60,000 images into memory:

```
string pixelFile = textBox1.Text;
string labelFile = textBox2.Text;
trainImages = LoadData(pixelFile, labelFile);
listBox1.Items.Add("MNIST training images loaded into memory");
```

The `Button2` click event handler displays the next image and updates the UI controls:

```
int nextIndex = int.Parse(textBox4.Text);
DigitImage currImage = trainImages[nextIndex];
int mag = int.Parse(comboBox1.SelectedItem.ToString());
Bitmap bitMap = MakeBitmap(currImage, mag);
pictureBox1.Image = bitMap;
textBox3.Text = textBox4.Text; // Update curr index
textBox4.Text = (nextIndex + 1).ToString(); // next index
textBox8.Text = textBox3.Text; // Random seed
listBox1.Items.Add("Curr image index = " + textBox3.Text +
    " label = " + currImage.Label);
```

You'll find the methods `LoadData` and `MakeBitmap` in the accompanying code download. Most of the distortion work is performed by the methods called by the `Button3` click event handler, which is presented in **Figure 2**.

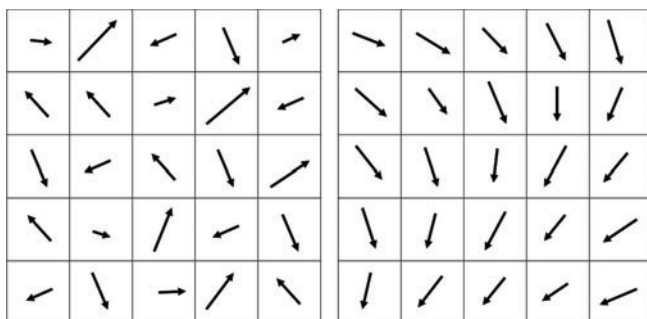


Figure 3 Random vs. Smoothed Displacement Fields

Method `Distort` calls methods `MakeKernel` (to create a smoothing matrix), `MakeDisplace` (directions and distances to deform each pixel in an image) and `Displace` (to actually deform the source image). Helper method `MakeDisplace` calls sub-helper `ApplyKernel` to smooth displacement values. Sub-helper method `ApplyKernel` calls sub-sub helper method `Pad`.

Elastic Deformation

The basic idea of distorting an image using elastic deformation is fairly simple. In the case of an MNIST image, you want to move each existing pixel slightly. But the details aren't so simple. A naive approach that moves each pixel independently leads to new images that appear broken up rather than stretched. For example, consider the conceptual images in **Figure 3**. The two images represent the distortion of a 5 x 5 section of an image. Each arrow indicates the direction and distance of a move by a corresponding pixel. The left image shows more or less random vectors, which would break up rather than distort an image. The right image shows vectors that are related to each other, leading to a stretched image.

So the trick is to displace each pixel in such a way that pixels close to each other move in a relatively similar, but not exactly the same, direction and distance.

So the trick is to displace each pixel in such a way that pixels close to each other move in a relatively similar, but not exactly the same, direction and distance. This can be accomplished using a matrix called a continuous Gaussian kernel. The overall idea is probably best explained using code. Consider this method in the demo:

```
private DigitImage Distort(DigitImage dImage, int kDim,
    double kStdDev, double intensity, int seed)
{
    double[][] kernel = MakeKernel(kDim, kStdDev);
    double[][] xField = MakeDisplace(dImage.width, dImage.height,
        seed, kernel, intensity);
    double[][] yField = MakeDisplace(dImage.width, dImage.height,
        seed + 1, kernel, intensity);
    byte[][] newPixels = Displace(dImage.pixels, xField, yField);
    return new DigitImage(dImage.width, dImage.height,
        newPixels, dImage.Label);
}
```

Method `Distort` accepts a `DigitImage` object and four numeric parameters related to a kernel. Type `DigitImage` is a program-defined class that represents the 28 x 28 bytes that make up the pixels of an MNIST image. The method first creates a kernel where `kDim` is the size of the kernel and `kStdDev` is a value that influences how similar the displacement of pixels will be.

To displace a pixel, it's necessary to know how far to move in the left-right direction, and in the up-down direction. This information is stored in arrays `xField` and `yField`, respectively, and is computed using helper method `MakeDisplace`. Helper method `Displace` accepts the pixel values of a `DigitImage` image and uses the displacement

Figure 4 Method MakeKernel

```
private static double[][] MakeKernel(int dim, double sd)
{
    if (dim % 2 == 0)
        throw new Exception("kernel dim must be odd");

    double[][] result = new double[dim][];
    for (int i = 0; i < dim; ++i)
        result[i] = new double[dim];

    int center = dim / 2; // Note truncation
    double coef = 1.0 / (2.0 * Math.PI * (sd * sd));
    double denom = 2.0 * (sd * sd);
    double sum = 0.0; // For more accurate normalization
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            int x = Math.Abs(center - j);
            int y = Math.Abs(center - i);
            double num = -1.0 * ((x * x) + (y * y));
            double z = coef * Math.Exp(num / denom);
            result[i][j] = z;
            sum += z;
        }
    }

    for (int i = 0; i < dim; ++i)
        for (int j = 0; j < dim; ++j)
            result[i][j] = result[i][j] / sum;

    return result;
}
```

fields to generate new pixel values. The new pixel values are then fed to a `DigitImage` constructor, yielding a new, distorted image. To summarize, to distort an image, you create a kernel. The kernel is used to generate x and y direction fields that are related rather than independent. The direction fields are applied to a source image to produce a distorted version of that image.

Gaussian Kernels

A continuous Gaussian kernel is a matrix of values that sum to 1.0; has the largest value in the center; and is radially symmetric. Here is a 5 x 5 Gaussian kernel with a standard deviation of 1.0:

0.0030	0.0133	0.0219	0.0133	0.0030
0.0133	0.0596	0.0983	0.0596	0.0133
0.0219	0.0983	0.1621	0.0983	0.0219
0.0133	0.0596	0.0983	0.0596	0.0133
0.0030	0.0133	0.0219	0.0133	0.0030

Notice that values near each other are different but similar. The standard deviation value determines how close the kernel values are. A larger standard deviation gives values that are closer together. For example, using a standard deviation of 1.5 gives a 5 x 5 kernel with first row values of:

0.0232	0.0338	0.0383	0.0338	0.0232
--------	--------	--------	--------	--------

This may seem odd at first because standard deviation is a measure of data spread and larger standard deviation values for a data set indicate a larger spread. But in the context of a Gaussian kernel, the standard deviation is used to generate the values and is not a measure of spread in the resulting kernel. The method used by the demo program to generate a Gaussian kernel is presented in **Figure 4**.

Generating Gaussian kernels can be a somewhat confusing task because there are many algorithm variations depending on how the kernel is intended to be used, and several variations of approximation techniques. The basic math definition for the values in a two-dimensional continuous Gaussian kernel is:

$$z = (1.0 / (2.0 * \pi^{*2})) * \exp(-(x^2 + y^2) / (2 * sd^2))$$

Here x and y are the x and y coordinates of a cell in the kernel relative to the center cell; pi is the math constant; exp is the exponential function; and sd is the specified standard deviation. The leading coefficient term of $1.0 / (2.0 * \pi^2)$ is actually a normalizing term for the one-dimensional version of a Gaussian function. But for 2D kernels, you want to sum all kernel preliminary values and then divide each preliminary value by the sum so that all final values will add to 1.0 (subject to rounding error). In **Figure 4**, this final normalization is accomplished using the variable named sum. Therefore, the variable named coef is redundant and can be omitted from the code; variable coef was included here because most research papers describe kernels using the coefficient term.

Displacement Fields

To distort an image, each pixel must be moved (virtually, not literally) a certain distance to the left or right, and up or down. Method `MakeDisplace` is defined in **Figure 5**. Method `MakeDisplace` returns an array-of-arrays-style matrix that corresponds to one-half of the conceptual matrix in **Figure 3**. That is, the values in the cells of the return matrix correspond to a direction and magnitude of a pixel move in either the x-direction or the y-direction. Because the size of an MNIST image is 28 x 28 pixels, the return matrix of `MakeDisplace` will also be 28 x 28.

Applying a kernel to a matrix
of displacements and then
using the resulting smoothed
displacements to generate new
pixel values is rather tricky.

Method `MakeDisplace` generates a matrix with initial random values between -1 and +1. Helper `ApplyKernel` smooths the random values as suggested by **Figure 3**. The smoothed values are essentially direction components with distance between 0 and 1.

Figure 5 Making a Displacement Field

```
private static double[][] MakeDisplace(int width, int height, int seed,
double[][] kernel, double intensity)
{
    double[][] dField = new double[height][];
    for (int i = 0; i < dField.Length; ++i)
        dField[i] = new double[width];

    Random rnd = new Random(seed);

    for (int i = 0; i < dField.Length; ++i)
        for (int j = 0; j < dField[i].Length; ++j)
            dField[i][j] = 2.0 * rnd.NextDouble() - 1.0;

    dField = ApplyKernel(dField, kernel); // Smooth

    for (int i = 0; i < dField.Length; ++i)
        for (int j = 0; j < dField[i].Length; ++j)
            dField[i][j] *= intensity;

    return dField;
}
```

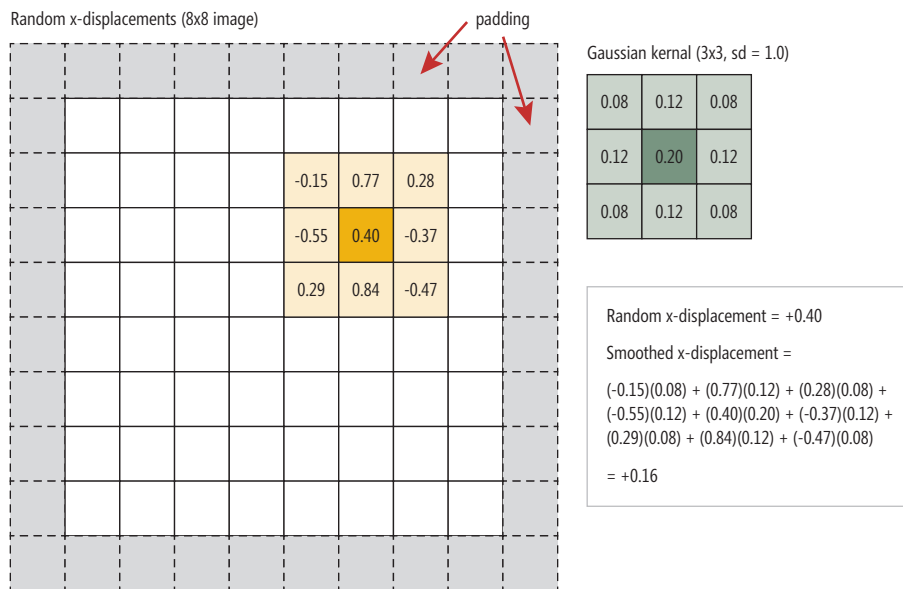


Figure 6 Applying a Kernel to a Displacement Matrix

Then all the values are multiplied by an intensity parameter to increase the stretch distance.

Applying a Kernel and a Displacement

Applying a kernel to a matrix of displacements and then using the resulting smoothed displacements to generate new pixel values is rather tricky. The first part of the process is illustrated in **Figure 6**. The left part of the figure represents the preliminary random displacement values between -1 and +1 in the x direction for an 8 x 8 image.

There are several ways to deal with the edge issue. One way is to pad the preliminary displacement matrix with dummy rows and columns. The number of rows or columns to pad will be equal to one-half (using integer truncation) of the dimension of the kernel.

The value at row [3], column [6] (0.40) is being smoothed using a 3 x 3 kernel. The new displacement is a weighted average of the current value and the values of the eight closest neighbors. Because each new displacement value is in essence an average of its neighbors, the net effect is to generate values that are related to each other.

After smoothing, the displacement values are multiplied by an intensity factor (sometimes called alpha in research literature). For example, if the intensity factor is 20, then the final x-displacement in **Figure 6** for the image pixel at (3, 6) would be $0.16 * 20 = +3.20$. There would be a similar y-displacements matrix. Suppose the final value at (3, 6) in the y-displacements matrix was -1.50. The values +3.20 and -1.50 that correspond to the pixel at (3, 6) are now applied to the source image to produce a distorted image, but not in an entirely obvious way.

First, lower and upper boundaries are determined. For the +3.20 x-displacement, these are 3 and 4. For the -1.50 y-displacement, they're -2 and -1. The four boundaries generate four (x, y) displacement pairs: (3, -2), (3, -1), (4, -2), (4, -1). Recall these values correspond

to the original image pixel value at indices (3, 6). Combining the pixel indices with the four displacement pairs generates four indices pairs: (6, 4), (6, 5), (7, 4), (7, 5). Finally, the pixel value for the distorted image at (3, 6) is the average of the original pixel values at indices (6, 4), (5, 3), (7, 4) and (7, 5).

Because of the geometry used, it's most common to restrict kernels to an odd dimension such as 3, 5 and so on. Notice that there'd be a problem trying to smooth any preliminary displacement values near the edge of the displacements matrix because the kernel would extend, so to speak, beyond the edge of the matrix. There are several ways to deal with the edge issue. One way is to pad the preliminary displacement matrix with dummy rows and columns. The number of rows or columns to pad will be equal to one-half (using integer truncation) of the dimension of the kernel.

Wrapping Up

The image elastic deformation process described in this article is just one of many possible approaches. Most, but not all, of the distortion algorithm presented here was adapted from the research article, "Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis," which is available online at bit.ly/REzsnM.

The demo program generates deformed images in order to create additional training data for an image-recognition system. If you're actually training an image-recognition system you can refactor the demo code to generate new training data on the fly, or you can refactor the code to generate and then save the distorted images as a text or binary file. ■

Dr. James McCaffrey works for Microsoft Research in Redmond, Wash. He has worked on several Microsoft products including Internet Explorer and Bing. Reach him at jammc@microsoft.com.

THANKS to the following technical expert for reviewing this article:
Wolf Kienle (Microsoft Research)



Fun with C#

Learning a new language can help developers bring new insights and new approaches to writing code in other languages, a la C#. My personal preference here is F#, because I'm an F# MVP these days. While I briefly touched on functional programming in an earlier column (bit.ly/1PaLNr), I want to look at a new language here.

When doing this, it's likely the code will eventually need to be written in C# (I'll do that in the next installment). But it can still be helpful to code it in F# for three reasons:

1. F# can sometimes solve problems like this more easily than C#.
2. Thinking about a problem in a different language can often help clarify the solution before rewriting it in C#.
3. F# is a .NET language like its cousin C#. So you could conceivably solve it in F#, then compile it into a .NET assembly and simply call in to it from C#. (Depending on the complexity of the calculation/algorithm, it could actually be the more sane solution.)

Look at the Problem

Consider a simple problem for this type of solution. Imagine you're working on Speedy, an application for managing personal finance. As part of the app, you need to "reconcile" transactions you've found online with the transactions a user has entered into the app. The goal here is to work through two lists of mostly identical data, and match the identical elements. What you do with those unmatched elements isn't yet unspecified, but you need to capture them.

Years ago, I did some contracting for the "intuitive" company that makes what was then the most popular PC application for managing your banking. This was an actual problem I had to work on there. It was specifically for the checking account register view after downloading a user's transactions as known by the bank. I had to reconcile those online transactions with ones the user had already entered into the app, and then ask the user about any transactions that didn't match.

Each transaction consists of an amount, a transaction date and a descriptive "comment." Here's the rub: The dates didn't always match and neither did the comments.

This means the only real credible data I could compare was the transaction amount. Fortunately, it's pretty rare within a given month that two transactions will be absolutely identical to the penny. So this is a "good enough" solution. I'll go back and confirm they are in fact a legit match. Just to complicate things, the two incoming lists don't have to match in length.

An F#ing Solution

There are principles about functional languages that dominate how you "think functionally." In this case, one of the first is they pre-

fer recursion over iteration. In other words, while the classically trained developer will immediately want to stand up a couple of nested for loops, the functional programmer will want to recurse.

Here, I'll take the local list of transactions and the remote list of transactions. I'll go through the first element of each list. If they match, I'll peel those two off their respective lists, mash them together into the result list, and recursively call again the remainder of the local and remote lists. Look at the type definitions for what I'm working with:

```
type Transaction =  
{  
    amount : float32;  
    date : System.DateTime;  
    comment : string  
}  
  
type Register =  
    | RegEntry of Transaction * Transaction
```

In simple terms, I'm defining two types. One is a record type, which is really an object without some of the traditional object notation. The other is a discriminated union type, which is really an object/class graph in disguise. I won't get into the depths of the F# syntax here. There are plenty of other resources out there for that, including my book, "Professional F# 2.0" (Wrox, 2010).

Suffice it to say, these are the input types and output types, respectively. The reason I chose a discriminated union for the

Figure 1 Create a Console Algorithm with F# REPL

```
[<EntryPoint>]  
let main argv =  
  
    let test1 =  
        let local = [ { amount = 20.00f;  
                        date = System.DateTime.Now;  
                        comment = "ATM Withdrawal" } ]  
        let remote = [ { amount = 20.00f;  
                        date = System.DateTime.Now;  
                        comment = "ATM Withdrawal" } ]  
        let register = reconcile local remote  
        Debug.Assert(register.Length = 1, "Matches should have come back with one item")  
  
    let test2 =  
        let local = [ { amount = 20.00f;  
                        date = System.DateTime.Now;  
                        comment = "ATM Withdrawal" };  
                      { amount = 40.00f;  
                        date = System.DateTime.Now;  
                        comment = "ATM Withdrawal" } ]  
        let remote = [ { amount = 20.00f;  
                        date = System.DateTime.Now;  
                        comment = "ATM Withdrawal" } ]  
        let register = reconcile local remote  
        Debug.Assert(register.Length = 1, "Register should have come back  
with one item")  
  
    0 // Return an integer exit code
```

Figure 2 Use F# Pattern Matching for a Recursive Solution

```
let reconcile (local : Transaction list) (remote : Transaction list) :
  Register list =
  let rec reconcileInternal outputSoFar local remote =
    match (local, remote) with
    | [], _ -> outputSoFar
    | _, [] -> outputSoFar
    | loc :: locTail, rem :: remTail ->
      match (loc.amount, rem.amount) with
      | (locAmt, remAmt) when locAmt = remAmt ->
        reconcileInternal (RegEntry(loc, rem) :: outputSoFar) locTail remTail
      | (locAmt, remAmt) when locAmt < remAmt ->
        reconcileInternal outputSoFar locTail remTail
      | (locAmt, remAmt) when remAmt > locAmt ->
        reconcileInternal outputSoFar locTail remTail
      | (_, _) ->
        failWith("How is this possible?")
  reconcileInternal [] local remote
```

result will soon become apparent. Given these two type definitions, it's pretty easy to define the outer skeleton of what I want this function to look like:

```
let reconcile (local : Transaction list) (remote : Transaction list) :
  Register list =
  []
```

Remember, in F# parlance, the type descriptors come after the name. So this is declaring a function that takes two Transaction lists and returns a list of Register items. As written, it stubs out to return an empty list ("[]"). This is good, because I can now stub out a few functions to test—Test-Driven Development (TDD) style—in a plain vanilla normal F# console application.

I can and should write these in a unit test framework for now, but I can accomplish essentially the same thing using System.Diagnostics.Debug.Assert and locally nested functions inside of main. Others may prefer to work with the F# REPL, either in Visual Studio or at the command line, as shown in **Figure 1**.

Given that I have a basic test scaffold in place, I'll attack the recursive solution, as you can see in **Figure 2**.

As you'll notice, this makes pretty heavy use of F# pattern matching. This is conceptually similar to the C# switch block (in the same way a kitten is conceptually similar to a saber-toothed tiger). First, I define a local recursive (rec) function that's basically the same

signature as the outer function. There's an additional parameter to carry the matched results so far.

Within that, the first match block examines both the local and remote lists. The first match clause ([],_) says if the local list is empty, I don't care what the remote list is (the underscore is a wildcard) because I'm done. So just return the results obtained so far. The same goes for the second match clause (_, []).

The meat of the whole thing comes in the last match clause. This extracts the head of the local list and binds it to the value loc, puts the rest of the list into locTail, does the same for remote into rem and remTail, and then matches again. This time, I extract the amount fields from each of the two items peeled off out of the lists, and bind them into the local variables locAmt and remAmt.

For each of these match clauses, I'll recursively invoke reconcileInternal. The key difference is what I do with the outputSoFar list before I recurse. If the locAmt and remAmt are the same, it's a match, so I prepend a new RegEntry into the outputSoFar list before recursing. In any other case, I just ignore them and recurse. The result will be a list of RegEntry items, and that's what's returned to the caller.

Extend the Idea

Suppose I can't just ignore those mismatched items. I need to put an item into the resulting list that says it was an unmatched local transaction or an unmatched remote transaction. The core algorithm still holds, I just add new items into the Register discriminated union to hold each of those possibilities, and append them into the list before recursing, as shown in **Figure 3**.

Now the results will be a complete list, with MissingLocal or MissingRemote entries for each Transaction that doesn't have a corresponding pair. Actually, this isn't quite true. If the two lists are mismatched in length, like my test2 case earlier, the leftover items won't be given "Missing" entries.

By taking F# as the "conceptualizing" language instead of C# and using functional programming principles, this became a pretty quick solution. F# uses extensive type inference, so in many cases while fleshing out the code, I didn't have to determine the actual types for the parameters and return ahead of time. Recursive functions in F# will often need a type annotation to define the return type. I got away without it here because it could infer from the return type given on the outer, enclosing function.

In some cases, I could just compile this into an assembly and hand it to the C# developers. For a lot of shops, though, that's not going to fly. So next time, I'll convert this to C#. The boss will never know that this code actually started life as F# code.

Happy coding! ■

Figure 3 Add New Items to the Register

```
type Register =
  | RegEntry of Transaction * Transaction
  | MissingRemote of Transaction
  | MissingLocal of Transaction

let reconcile (local : Transaction list) (remote : Transaction list) :
  Register list =
  let rec reconcileInternal outputSoFar local remote =
    match (local, remote) with
    | [], _ -> outputSoFar
    | _, [] -> outputSoFar
    | loc :: locTail, rem :: remTail ->
      match (loc.amount, rem.amount) with
      | (locAmt, remAmt) when locAmt = remAmt ->
        reconcileInternal (RegEntry(loc, rem) :: outputSoFar) locTail remTail
      | (locAmt, remAmt) when locAmt < remAmt ->
        reconcileInternal (MissingRemote(loc) :: outputSoFar) locTail remote
      | (locAmt, remAmt) when locAmt > remAmt ->
        reconcileInternal (MissingLocal(rem) :: outputSoFar) local remTail
      | _ ->
        failWith "How is this possible?"
  reconcileInternal [] local remote
```

TED NEWARD is the CTO at iTrellis, a consulting services company. He has written more than 100 articles and authored and coauthored a dozen books, including "Professional F# 2.0" (Wrox, 2010). He's a C# MVP and speaks at conferences around the world. He consults and mentors regularly—reach him at ted@tedneward.com or ted@itrellis.com if you're interested in having him come work with your team, and read his blog at blogs.tedneward.com.

THANKS to the following Microsoft technical expert for reviewing this article:
Lincoln Atkinson



Authentication and User Identity in Windows Store Apps

An overwhelming number of apps and Web sites now require some sort of login or credentials. One reason is that many of them customize the experience based on saved user preferences. Because so many sites and apps require user sign in, you need to make it easy for users to do so. In this article, you'll learn the basics of app authentication on the Windows platform by enabling Microsoft Account or third-party login, as well as supplemental authentication methods such as smart card or biometric authentication.

Modern App Authentication and User Identity

More often than not, gathering user credentials is more of a pain to the user than an enriching experience. Therefore, when you must store user credentials, you want to do so in a way that keeps things secure, yet maintains ease of use.

Using an independent credential provider such as Microsoft Account (formerly Windows Live ID), Facebook, or OAuth is a great way to lean on a trusted entity to manage and store user information. This still provides users with a persistent, individualized UI. Also, let's face it—a user simply doesn't remember all his passwords, especially in sites and apps he uses infrequently. Asking a user to create and remember another set of credentials just adds friction to his experience. That alone can cause a drop in your app store ratings, which is the opposite effect of what you want.

On the developer side, storing user credentials means you have to write more code, tests and bug fixes than you would if you were using something such as a Facebook login. There's also the potential for security breaches when storing login data in local databases. You can alleviate stress for both users and yourself by using a trusted entity such as Microsoft, Facebook, Twitter, OAuth or any credible third-party credential service. The service stores user credentials

and private information using proper algorithms so it remains secure. All you have to do is make some calls to the API, and that does all the heavy lifting.

Using Microsoft Account

When using Microsoft Account, users expect the login experience to be similar and consistent with Windows login. Instead of creating your own UI controls for collecting the information, you should use those that come with the Microsoft Account APIs. Fortunately, that also makes it easier to code. Coding login information is a boring task you can and should abstract with APIs such as the Microsoft Account APIs.

Using Microsoft Account
authentication gives you a
fluid sign-on experience to all
Microsoft Live services.

If you're going to require a login, show the login UI upon starting the app. If the user can stay signed in between sessions or doesn't have to log in, add that to the Settings. Also, don't forget to make sure the user can sign out. That's so easy to forget, as many users prefer to stay signed in to favorite Web sites and apps.

This can be an issue if someone needs to borrow a phone or device. If he signs in and can't sign out, the next person could potentially use the app and masquerade as the first one. On the other hand, the first person may never be able to share his device due to not being able to sign out in the first place.

Be sure to keep sign-in status where a user can readily see it from all screens in your app. This doesn't have to be a boring text update. You can show the user's avatar or perhaps some photos from his OneDrive photo storage (if using Microsoft Account). Be as creative as you want, as long as the user can see and change his status with ease. A common technique is to show the user's avatar thumbnail in the top right corner of the app.

Microsoft Account Authentication

Using Microsoft Account authentication gives you a fluid sign-on experience to all Microsoft Live services, such as OneDrive,

Figure 1 This Code Creates the Account Settings Flyout

```
<SettingsFlyout
  x:Class="App.Account"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:App"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d"
  IconSource="Assets/SmallLogo.png"
  Title="Account"
  d:DesignWidth="346">

  <StackPanel x:Name="accountSettings">
    <Button x:Name="SignInButton" Click="SignInClick" Content="Sign in" />
  </StackPanel>
</SettingsFlyout>
```


Outlook.com, an MSDN subscription, Xbox, and even Windows 8 devices and apps.

To be able to use services that may contain sensitive user information, you must register the app with the Microsoft Dev Center online (bit.ly/1egpvHx). Doing so lets you use Microsoft Account services in your app so you can connect users to various resources.

Because you're working with services that access personal data, you have to provide a clear privacy statement accessible within the app. If you need help writing a privacy policy, you can use myapppolicy.com or w8privacy.azurewebsites.net to help you build one. Once you've established a privacy policy, you can display it with a Settings flyout.

With Web Authentication Broker, you don't have to deal with the hassles of securely storing user information.

Finally, you need to deliver code to sign people in and out. You can do this with a Flyout control that contains a sign-in button (this changes to a sign-out button when signed in). The buttons launch dialogs to perform the corresponding operations of signing in or out. **Figure 1** shows the XAML that creates an Account Settings Flyout.

As you can see, it's a control with a button. The button is what the user will use to launch the login interface. It's a customary practice to put a sign-in button in a Flyout. To learn more about creating Settings Flyouts, see "Mastering Controls and Settings in Windows Store Apps" (bit.ly/1hy7fk2). The code that signs users into your app from the click event of the sign-in button should look something like this:

```
private async void SignInButton_Click(object sender, RoutedEventArgs e)
{
    LiveAuthClient authClient = new LiveAuthClient();
    LiveLoginResult authResult =
        await authClient.LoginAsync(new List<string>() { "wl.signin", "wl.basic" });
    if (authResult.Status == LiveConnectSessionStatus.Connected)
    {
        // Perform post authentication duties
    }
}
```

This displays the Microsoft account sign-in screen (often called the login consent page), if the user hasn't already signed in (see **Figure 2**). As you can see, it doesn't take much code to make the login process happen. You can use the Session property of the LiveAuthClient to query for session status throughout the app.

You can perform the same activities with HTML and JavaScript, but the steps are slightly different. First, reference the Windows Live SDK with the Add Reference command. Then, as is customary in HTML development, reference the scripts with the `<script>` tag in your HTML document, like this:

```
<script src="//LiveSDKHTML/js/wl.js"></script>
```

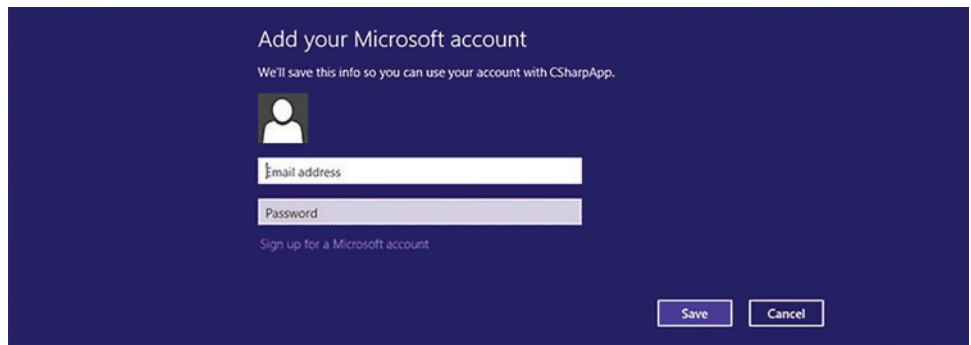


Figure 2 The Microsoft Account Sign-in Screen

However, you might want to use the debug version during development. In that case, you can use the following reference:

```
<script src="//js.live.net/v5.0/wl.debug.js"></script>
```

Figure 3 contains the WinJS code that signs in a user. You need to call `WL.init` and `WL.Event.subscribe` so you can initialize the Microsoft Account, as well as subscribe to an event that happens upon login. XAML apps don't require this initialization. In the login event, you can display connection status or do whatever is required just after logging in.

The code in **Figure 3** does the same thing as its XAML counterpart—it actually logs in a user. Regardless of the language you use, the `wl.login` method requires you to pass in a scope. A scope is a permission level, or a scope of where and what an app can do on behalf of the user. As you might imagine, scopes are user-controlled. You can try to do anything you want while posing as the user, but the Windows Runtime will force you to obtain user consent before proceeding.

Web Authentication Broker (OAuth)

Using a centralized and objective third-party credential service such as Microsoft Account, Facebook, Twitter, LinkedIn, OAuth and so on is a good authentication strategy. Users are more likely to remember their passwords for sites they visit frequently or prefer more. Using a reputable authenticator also means less infrastructure code on which you have to focus, and more time for business logic or UI work.

It's a customary practice to put a sign-in button in a Flyout.

Web Authentication Broker is one way to easily perform authentication. With Web Authentication Broker, you don't have to deal with the hassles of securely storing user information. Instead, you properly encrypt all personal information. Web Authentication Broker is a liaison between an app and an authentication provider. It also enables single sign-on across multiple apps.

Call the Web Authentication Broker in code to retrieve and display a dialog box or Web page from a specific authentication service such as Facebook (see **Figure 4**). The dialog is the same sign-in screen the user would normally see when logging into the provider's Web site or app. In most cases, you can customize it so it feels like a natural part of your app.

Smart Card Authentication

Microsoft has long used smart cards as a way to authenticate VPN connections to the company's networks, as well as provide secure access to buildings and other corporate resources. Now you can easily incorporate smart cards into your software. The `Windows.Devices.SmartCards` namespace contains classes such as `SmartCard` and `SmartCardConnection` that let you write code to authenticate and manage smart cards. While smart cards may seem overly complicated, they're not. If you wanted to verify what cards are currently located in the reader, for example, you can write code similar to the following:

```
string selector = SmartCardReader.GetDeviceSelector();
DeviceInformationCollection devices = await DeviceInformation.
FindAllAsync(selector);

foreach (DeviceInformation device in devices)
{
    SmartCardReader reader =
        await SmartCardReader.FromIdAsync(device.Id);

    IReadOnlyList<SmartCard> cards = await reader.FindAllCardsAsync();
}
```

Notice that `Windows.Devices.DeviceInformation` obtains information about each device. In Windows, this is a standard way to query for hardware. The Windows Runtime exposes many of the objects that communicate with hardware that were previously unavailable to .NET developers.

Biometric Authentication

Although it's unlikely you could use another form of authentication besides the standard username/password technique because many devices and machines don't support the hardware, you can use a stored and registered fingerprint to read software. It's a great feature and quite helpful to those who may have accessibility needs.

Consider the Lenovo Carbon Touch X1 laptop computer. This model has a fingerprint reader built into the laptop near the keyboard. With Windows 8 biometric authentication, you can use

Figure 3 The WinJS Code That Signs the User In

```
(function () {
    "use strict";

    WL.init();
    WL.Event.subscribe("auth.login", loginComplete);

    document.querySelector("#SignInButton").addEventListener("click",
        function () {
            login();
        });

    function loginComplete() {
        WinJS.log("User has signed in!")
    }

    function login() {
        var session = WL.getSession();
        if (session) {
            WinJS.log("You are already signed in!")
        }
        else {
            WL.login({ scope: "wl.basic" }).then(
                function () {
                    // Perform post authentication duties
                },
                function (response) {
                    WinJS.log("Could not connect, status = " + response.status);
                });
        }
    }
})();
```

Figure 4 Signing in with Web Authentication Broker

```
async public Task<string> WebAuthenticate(){

    string startURL =
        "https://<providerendpoint>?client_
        id=<clientid>&scope=<scopes>&response_type=token";
    string endURL = "http://<appendpoint>";

    System.Uri startURI = new System.Uri(
        "https://<providerendpoint>?client_
        id=<clientid>&scope=<scopes>&response_type=token");
    System.Uri endURI = new System.Uri(
        "http://<appendpoint>");

    string result;

    try
    {
        var webAuthenticationResult =
            await WebAuthenticationBroker.AuthenticateAsync(
                WebAuthenticationOptions.None,
                startURI,
                endURI);

        switch (webAuthenticationResult.ResponseStatus)
        {
            case WebAuthenticationStatus.Success:
                result = webAuthenticationResult.ResponseData.ToString();
                break;
            case WebAuthenticationStatus.ErrorHttp:
                result = webAuthenticationResult.ResponseErrorDetail.ToString();
                break;
            default:
                result = webAuthenticationResult.ResponseData.ToString();
                break;
        }
    }
    catch (Exception ex)
    {
        result = ex.Message;
    }
    return result;
}
```

Figure 5 Fingerprint Authentication

```
var availability = await UserConsentVerifier.CheckAvailabilityAsync();

if (UserConsentVerifierAvailability.Available) {
    var consentResult = await UserConsentVerifier.
    RequestVerificationAsync(userMessage);
}

var consentResult = await UserConsentVerifier.RequestVerificationAsync(userMessage);

switch (consentResult)
{
    case UserConsentVerificationResult.Verified:
        returnMessage = "Fingerprint verified.";
        break;
    case UserConsentVerificationResult.DeviceBusy:
        returnMessage = "Biometric device is busy.";
        break;
    case UserConsentVerificationResult.DeviceNotPresent:
        returnMessage = "No biometric device found.";
        break;
    case UserConsentVerificationResult.NotConfiguredForUser:
        returnMessage = "The user has no fingerprints registered.";
        break;
    case UserConsentVerificationResult.RetriesExhausted:
        returnMessage = "Too many failed attempts.";
        break;
    case UserConsentVerificationResult.Canceled:
        returnMessage = "Fingerprint authentication canceled.";
        break;
    default:
        returnMessage = "Fingerprint authentication is currently unavailable.";
        break;
}
```

this type of authentication. **Figure 5** shows the code you would use for biometric authentication.

The code in **Figure 5** displays a modal dialog much like the one in **Figure 2**, except it will prompt the user to scan their finger on the biometric device during the call to `CheckAvailabilityAsync`. Once the user scans his finger, you can query the consent result to see if it's a verifiable scan.

Signing Out

Using Microsoft Account, you can deliver a rich and consistent experience that integrates seamlessly into Windows 8. Now you have a variety of authentication providers, which you should use before building your own. Whether you use Twitter, Facebook, Microsoft, or others, they've already addressed the challenges and issues that come with securely managing private user information. Those providers also keep security up-to-date. This makes reputable third-party validation the best choice for modern app authentication.

Unless it's an absolute must, don't create your own algorithms to store private user information. Use a service such as Credential Locker if you need to retain that information. You can hang onto user credentials safely with Credential Locker. This gives the app a way to automatically sign in a user across app sessions, because her credentials are in a vault in the cloud, which enables roaming.

While smart cards
may seem overly
complicated,
they're not.

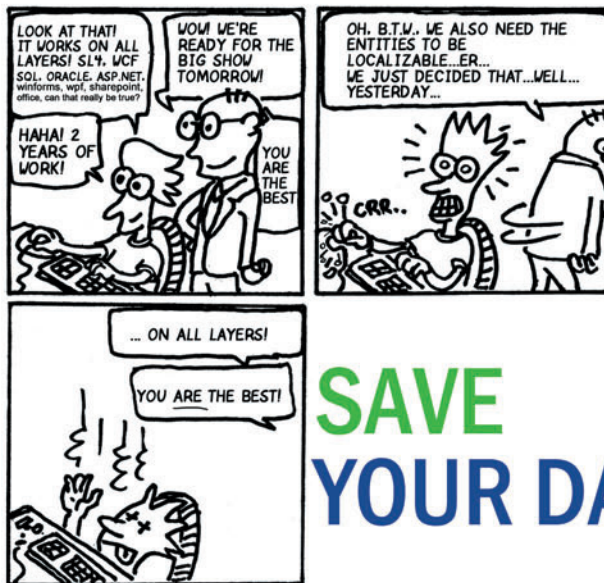
Roaming is a must-have feature when your app runs on multiple devices (such as a tablet and phone). Using Credential Locker, you can securely lock down credentials, yet pass them between versions of your app, as well as social networking or other external sites. Moreover, you don't have to maintain infrastructure to store user information. You can find more details about the Credential Locker at bit.ly/1qFxmG.

With augmented authentication like biometrics, you're sure to have the highest-rated apps in the store. The UI is modern, easy to use, and has all the bells and whistles. Not all devices will have the exact same

authentication peripherals, so don't rely on something like biometrics as the sole means of authentication. ■

RACHEL APPEL is a consultant, author, mentor and former Microsoft employee with more than 20 years of experience in the IT industry. She speaks at top industry conferences such as Visual Studio Live!, DevConnections, MIX and more. Her expertise lies within developing solutions that align business and technology focusing on the Microsoft dev stack and open Web. For more about Appel, visit her Web site at rachelappel.com.

THANKS to the following Microsoft technical expert for reviewing this article:
Frank La Vigne



Use CodeFluent Entities

CodeFluent Entities is a unique product integrated into Visual Studio that allows you to generate database scripts, code (C#, VB), web services and UIs.

"I recently spent a week attending a course on Entity Framework but CodeFluent Entities provides so much more and is decidedly easier to understand and implement" *

Peter Stanford - Artefaction - Australia

* Source : <http://visualstudiogallery.msdn.microsoft.com/B6299BBF-1EF1-436D-B618-66E8C16AB410>

To get a license worth \$399 for free
Go to www.softfluent.com/forms/msdn-2014



More information: www.softfluent.com Contact us: info@softfluent.com



Breaking the Z Barrier with Direct2D Effects

As children, we learn how to paint before we learn how to read and write, and we undoubtedly glean a few lessons from the experience. We discover the temporal process of painting is reflected in the layering of paint on the canvas. What we paint earlier may be partially covered up and obscured by what we paint later.

For that reason, even someone completely unfamiliar with the mechanics of computer graphics can probably guess how the image in **Figure 1** was rendered: Obviously, the background was colored gray first; next came the blue triangle, followed by the green, and last by the red, which is in front of everything else. It's no surprise the process of rendering figures from rear to front is known as the "painter's algorithm."

The three triangles in **Figure 1** might also be pieces of colored construction paper arranged in a stack. If you were to add more and more triangles to the stack, they would build up into a pile, and what started out as a two-dimensional surface would acquire a third dimension.

Even in 2D graphics, there's a rudimentary concept of a Z axis—a virtual space orthogonal to the two-dimensional screen or canvas. The layering of flat 2D objects is governed by the "Z order" of the figures. In XAML-based environments, for example, the `Canvas.ZIndex` attached property determines what elements seem to sit on top of others, but it really just controls the order the elements are rendered on the screen.

The problem is this: In 2D graphics, a Z index always applies to the *entire* figure. You can't use this type of Z ordering to draw three figures like those in **Figure 2**, with the first on top of the second, the second on top of the third, but the third on top of the first.

That change in one corner of one triangle seems slight, but what a world of difference it represents! The image in **Figure 2** might be easy with construction paper, but not so easy with painting—either in real life or in 2D graphics programming. One of these triangles needs to be rendered in two parts with carefully calculated coordinates, or using clipping based on one of the other triangles.

Effects and the GPU

The rendering of **Figure 2** can be helped enormously by borrowing some concepts from the world of 3D graphics.

Such figures can't have uniform Z indices; instead, the figures must be allowed to have variable Z coordinates over their entire surfaces. The drawing process can then maintain a collection of Z coordinates (called a Z buffer or a depth buffer) that encompasses every pixel of the rendering surface. As each figure is rendered, the

Z coordinate of each pixel of the figure is compared with the corresponding Z coordinate in this depth buffer. If the pixel is on top of the Z coordinate in the depth buffer, the pixel is drawn and the new Z coordinate is stored in the depth buffer. If not, the pixel is ignored.

This sounds computationally costly—not only for the comparison itself but for the calculation of Z coordinates for every pixel of each graphical figure—and that's an accurate evaluation. That's why it's an ideal job to hand over to the parallel computational capabilities of the modern GPU.

The calculation of Z coordinates for every pixel of a figure is conceptually rather easy if the figure happens to be a triangle—and keep in mind every polygon can be decomposed into triangles. All that's necessary is to give each of the three vertices of the triangle a 3D coordinate point, and then any point within the triangle can be calculated as a weighted average of the three vertex coordinates. (This involves barycentric coordinates—not the only concept used in computer graphics developed by German mathematician August Ferdinand Möbius.)

That same interpolation process can shade the triangle, as well. If each vertex is assigned a specific color, then any pixel within that triangle is a weighted average of those three colors, as shown in **Figure 3**.

That type of color gradient is also an important feature of 3D programming because it allows triangles to be shaded to resemble curved surfaces. But it's not a type of gradient that's common in conventional 2D programming.

The image in **Figure 3** was created by a downloadable program called *ThreeTriangles* that runs under Windows 8.1 and Windows Phone 8.1. (The solution was created in Visual Studio 2013 Update 2 using the new Universal App template that allows sharing lots of code between Windows 8.1 and Windows Phone 8.1.)



Figure 1 Three Overlapping Triangles

Code download available at msdn.microsoft.com/magazine/msdnmag0714.

The graphics in the ThreeTriangles program are done entirely in Direct2D, using a feature of Direct2D called effects or (when you code them yourself) custom effects. Using custom effects you can get much closer to authentic 3D programming than otherwise possible with Direct2D.

When writing a custom effect for Direct2D, you acquire a privilege normally restricted to 3D programmers: You can write code that executes on the GPU. This code takes the form of little programs called shaders, which you write using a high-level shading language (HLSL) that resembles C. These shaders are compiled by Visual Studio into compiled shader object (.cso) files during the normal project build and then run on the GPU when the program executes.

Indeed, Direct2D effects are sometimes described as little more than wrappers for shaders! The custom effects are the only way you can use shaders within the context of Direct2D programming to achieve 3D-like images.

Three different types of shaders are available for use by Direct2D effects:

- A vertex shader that performs operations on vertices. Each triangle has three vertices. The vertices always involve a coordinate point but might include other information, such as color.
- A pixel shader that performs operations on all the pixels within these triangles. Any information supplied with the vertices is automatically interpolated over the surface of the triangle in preparation for the pixel shader.
- A compute shader that uses the GPU to perform heavy parallel processing. I won't be discussing the compute shader in this article.

The shaders used for Direct2D effects have somewhat different requirements than shaders associated with Direct3D programming, but many of the concepts are the same.

Built-in Effects and Custom Effects

Direct2D includes about 40 predefined built-in effects, which perform various image-processing manipulations on bitmaps, such as blur or sharpen, or various types of color manipulation.

Each of these built-in effects is identified by a class ID you use to create an effect of that type. For example, suppose you want to use the color-matrix effect, which allows specifying a transform to alter the colors in a bitmap. You'll probably declare an object of type `ID2D1Effect` as a private field in your rendering class:

```
Microsoft::WRL::ComPtr<ID2D1Effect> m_colorMatrixEffect;
```

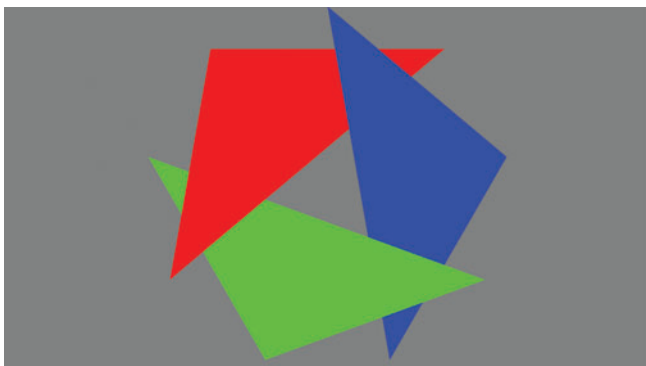


Figure 2 Mutually Overlapping Triangles

In the `CreateDeviceDependentResources` method, you can create this effect by referencing the documented class ID:

```
d2dContext->CreateEffect(  
    CLSID_D2D1ColorMatrix, &m_colorMatrixEffect);
```

At that point, you can call `SetInput` on the effect object to set a bitmap, and `SetValue` to specify a transform matrix. You render this color-shifted bitmap by calling:

```
d2dContext->DrawImage(m_colorMatrixEffect.Get());
```

All the built-in effects involve bitmap input, and one of the features of Direct2D effects is that you can chain them together to apply a series of effects to a bitmap.

Indeed, Direct2D effects are sometimes described as little more than “wrappers” for shaders!

If you're interested in writing your own custom effects, there's an invaluable Windows 8.1 Visual Studio solution called Direct2D Custom Image Effects Sample that includes three separate projects to demonstrate the three types of shaders available for Direct2D effects. All three programs require bitmaps as input.

Therefore, you'd be forgiven for assuming that Direct2D effects always perform operations on bitmap input. But this isn't so. The ThreeTriangles program that created the image in **Figure 3** doesn't require bitmap input.

You'd also be forgiven for assuming Direct2D effects involve just one type of shader. Certainly, the built-in effects seem to involve either a vertex shader or a pixel shader, but not both. However, the ThreeTriangles program is different in this respect, as well: It defines a custom effect that uses both a vertex shader and a pixel shader.

Register, Create, Draw

Because Direct2D effects are designed to be preregistered and created from a class ID, a custom effect needs to offer that same ability. The custom effect in the ThreeTriangles program is a class named `SimpleTriangleEffect`, which defines a static method for registering the class. This method is called by the constructor of the `ThreeTrianglesRenderer` class, but the effect could be registered anywhere in the program:

```
SimpleTriangleEffect::RegisterEffectAsync(d2dFactory)
```

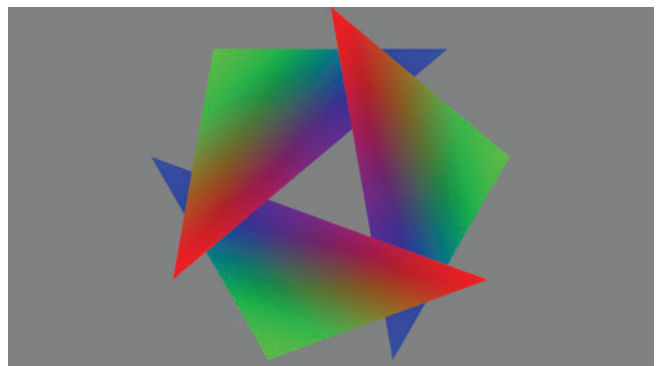


Figure 3 The ThreeTriangles Program Display

This registration method is asynchronous because it needs to load in the compiled shader files, and the only method provided for this purpose in the DirectXHelper class is ReadDataAsync.

Just like when using a built-in effect, the ThreeTrianglesRenderer class declares an ID2DIEffect object as a private field in its header file:

```
Microsoft::WRL::ComPtr<ID2DIEffect> m_simpleTriangleEffect;
```

The CreateDeviceDependentResources method creates the custom effect the same way as a built-in effect:

```
d2dContext->CreateEffect(
    CLSID_SimpleTriangleEffect, &m_simpleTriangleEffect)
```

The earlier registration of the custom effect associated that class ID with the effect.

The SimpleTriangleEffect has no input. (That's part of what makes it "simple"!)

The effect is rendered just like a built-in effect:

```
d2dContext->DrawImage(m_simpleTriangleEffect.Get());
```

Perhaps the simple use of this custom effect suggests some of the complexity within the effect class itself. A custom effect such as SimpleTriangleEffect must implement the ID2DIEffectImpl (effect implementation) interface. An effect can consist of multiple passes, which are called transforms, and each one is usually represented by an implementation of ID2DIDrawTransform. If a single class is used for both interfaces—which is the case with SimpleTriangleEffect—then it needs to implement IUnknown (three methods), ID2DIEffectImpl (three methods), ID2DITransformNode (one method), ID2DITransform (three methods), and ID2DIDrawTransform (1 method).

That's a considerable amount of overhead, in addition to some XML that identifies the effect and its author when the effect is first registered. Fortunately, for simple effects—and this one certainly qualifies—many of the effect methods can have fairly easy implementations. The most important jobs of the effect class involve loading and registering the compiled shader code (and associating the shaders with GUIDs for later reference), and defining a vertex buffer, which must also be associated with a GUID.

From Vertex Buffer ...

A vertex buffer is a collection of vertices assembled for processing. Each vertex always includes a 2D or 3D coordinate point, but usually other items as well. The data associated with each vertex and how it's organized is called the "layout" of the vertex buffer and, overall, the ThreeTriangles program defines three different—but equivalent—data types to describe this vertex layout.

The first representation of this vertex data is shown in **Figure 4**. This is a simple structure named Vertex that includes a 3D coordinate point and an RGB color. An array of these structures defines the three triangles displayed by the program. (This array is hardcoded in the required Initialize method of the SimpleTriangleEffect class; in a real program the effect class would allow an array of vertices to be input to the effect.)

The x and y values are based on sines and cosines of angles in increments of 40 degrees, with a radius of 1,000. However, notice that the z coordinates are all set between 0 and 1, so that the red vertices have a z value of 1, the green vertices are 0.5, and the blue vertices are 0. More on this a little later.

Following that array is another little array, but this one defines the vertex information in a more formal manner required for creating and registering the vertex buffer.

The vertex buffer and the vertex shader are both referenced in the SetDrawInfo method of SimpleTriangleEffect. Every time the effect is rendered, these nine vertices are passed to the vertex shader.

... To Vertex Shader ...

Figure 5 shows the vertex shader for the SimpleTriangleEffect. It consists of three structures and a function called main. The main function is called for every vertex in the vertex buffer; in this case, that's only nine vertices, but often there are many more.

Each of the three structures contains fields identified with an HLSL data type, a member name and uppercase semantics that identify the role of the particular field.

The structure named VertexShaderInput is the input to main, and it's the same as the layout of the vertex buffer you've just seen, but with HLSL data types for the 3D position and the RGB color.

The structure named VertexShaderOutput defines the output of main. The first two fields are required for Direct2D effects. (A third required field would be present if the effect involved an input bitmap.) The field I've called sceneSpaceOutput is based on the input coordinate. Some effects change that coordinate; this effect does not, and simply turns the 3D input coordinate into a 4D homogenous coordinate with a w value of 1:

```
output.sceneSpaceOutput = float4(input.position.xyz, 1);
```

The vertex shader output also includes a non-required field called color, which is simply set from the input color:

```
output.color = input.color;
```

The required output field I've called clipSpaceOutput describes each vertex coordinate in terms of normalized coordinates used in 3D. These coordinates are the same as coordinates generated

Figure 4 Vertex Definition in SimpleTriangleEffect

```
// Define Vertex for simple initialization
struct Vertex
{
    float x;
    float y;
    float z;
    float r;
    float g;
    float b;
};

// Each triangle has three points and three colors
static Vertex vertices [] =
{
    // Triangle 1
    { 0, -1000, 0.0f, 1, 0, 0 },
    { 985, -174, 0.5f, 0, 1, 0 },
    { 342, 940, 1.0f, 0, 0, 1 },

    // Triangle 2
    { 866, 500, 0.0f, 1, 0, 0 },
    { -342, 940, 0.5f, 0, 1, 0 },
    { -985, -174, 1.0f, 0, 0, 1 },

    // Triangle 3
    { -866, 500, 0.0f, 1, 0, 0 },
    { -643, -766, 0.5f, 0, 1, 0 },
    { 643, -766, 1.0f, 0, 0, 1 }
};

// Define layout for the effect
static const D2D1_INPUT_ELEMENT_DESC vertexLayout [] =
{
    { "MESH_POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0 },
    { "COLOR", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12 },
};
```


from the camera projection transforms I described in last month's installment of this column. In these clipSpaceOutput coordinates, x values range from -1 on the left of the screen to 1 on the right; y values range from -1 on the bottom to 1 at top; and z values range from 0 for coordinates closest to the viewer to 1 for coordinates furthest away. As the name of the field implies, these normalized coordinates are used for clipping the 3D scene to the screen.

To assist you in calculating these clipSpaceOutput coordinates, a third structure is automatically provided for you that I've called ClipSpaceTransforms. These are four numbers based on the pixel width and height of the screen, and any device context transforms that are in effect when DrawImage renders the effect.

However, the provided transforms are only for x and y coordinates, and that's why I defined z coordinates in the original vertex buffer to have values between 0 and 1. Another approach is to use an actual camera projection transform in the vertex shader (as I'll demonstrate in a future column).

These z values are also automatically used in a depth buffer so that pixels with lower z coordinates obscure pixels with higher z values. But this only occurs if the SetDrawInfo method in the

Figure 5 The SimpleTriangleEffectVertexShader.hlsl File

```
// Per-vertex data input to the vertex shader
struct VertexShaderInput
{
    float3 position : MESH_POSITION;
    float3 color : COLOR0;
};

// Per-vertex data output from the vertex shader
struct VertexShaderOutput
{
    float4 clipSpaceOutput : SV_POSITION;
    float4 sceneSpaceOutput : SCENE_POSITION;
    float3 color : COLOR0;
};

// Information provided for Direct2D vertex shaders
cbuffer ClipSpaceTransforms : register(b0)
{
    float2x1 sceneToOutputX;
    float2x1 sceneToOutputY;
}

// Called for each vertex
VertexShaderOutput main(VertexShaderInput input)
{
    // Output structure
    VertexShaderOutput output;

    // Append a 'w' value of 1 to the 3D input position
    output.sceneSpaceOutput = float4(input.position.xyz, 1);

    // Standard calculations
    output.clipSpaceOutput.x =
        output.sceneSpaceOutput.x * sceneToOutputX[0] +
        output.sceneSpaceOutput.w * sceneToOutputX[1];

    output.clipSpaceOutput.y =
        output.sceneSpaceOutput.y * sceneToOutputY[0] +
        output.sceneSpaceOutput.w * sceneToOutputY[1];

    output.clipSpaceOutput.z = output.sceneSpaceOutput.z;
    output.clipSpaceOutput.w = output.sceneSpaceOutput.w;

    // Transfer the color
    output.color = input.color;

    return output;
}
```

Figure 6 The SimpleTriangleEffectPixelShader.hlsl File

```
// Per-pixel data input to the pixel shader
struct PixelShaderInput
{
    float4 clipSpaceOutput : SV_POSITION;
    float4 sceneSpaceOutput : SCENE_POSITION;
    float3 color : COLOR0;
};

// Called for each pixel
float4 main(PixelShaderInput input) : SV_TARGET
{
    // Simply return color with opacity of 1
    return float4(input.color, 1);
}
```

effect class calls SetVertexProcessing with the D2D1_VERTEX_OPTIONS_USE_DEPTH_BUFFER flag. (This happens to also result in COM errors appearing in the Output window of Visual Studio while the program is running, but that also happens with the Microsoft sample Direct2D effect code.)

... To Pixel Shader

Every time the effect is rendered (and in the general case, that's at the frame rate of the video display), the vertex shader is called for every vertex in the vertex buffer, in this case nine times.

The output from the vertex shader has the same format as the input to the pixel shader. As you can see in the pixel shader in **Figure 6**, the PixelShaderInput structure is the same as the VertexShaderOutput structure in the vertex shader.

However, the pixel shader is called for every pixel in the triangles, and all the fields of the structure have been interpolated over the surface of that triangle. The main function in the pixel shader must return a four-component color that includes opacity, so the interpolated RGB color is simply modified by appending an opacity field. That color is output to the display.

Here's an interesting variation for the pixel shader: The z coordinates of the sceneSpaceOutput field range from 0 to 1, so it's possible to visualize the depth of each triangle by using that coordinate to construct a gray shade, and return it from the main method:

```
float z = input.sceneSpaceOutput.z;
return float4(z, z, z, 1);
```

Enhancements?

The SimpleTriangleEffect cuts some corners. It should be made more versatile by including a method to set the vertex input. Some other features wouldn't hurt either: The vertex shader is a great place to perform matrix transforms—such as rotation or camera transforms—because the matrix multiplications are executed on the GPU.

Few programmers are capable of resisting the temptation to implement code enhancements, particularly ones that turn a static image into an animated one. ■

CHARLES PETZOLD is a longtime contributor to MSDN Magazine and the author of *Programming Windows, 6th Edition* (Microsoft Press, 2013), a book about writing applications for Windows 8. His Web site is charlespetzold.com.

THANKS to the following Microsoft technical expert for reviewing this article:
Doug Erickson



Windows XP: An Old Soldier Fades Away

I'm writing this column the week after Microsoft officially discontinued support for Windows XP. About 28 percent of all computers worldwide still run it. Here, in my not especially Luddite town in Massachusetts, the public library, my kids' school and my bank all run it. The British and Dutch governments still run it, as does the U.S. Internal Revenue Service—our tax dollars at work. Microsoft just can't kill this thing off, no matter how hard it tries and how much the company wishes it could.

I remember the release of XP in 2001. Developers rejoiced that the consumer and industrial versions of Windows had finally merged. We no longer had to develop two versions of all our applications, one for Windows 95/98 and one for Windows NT. Life was improving.

We'll probably keep bumping into Windows XP now and again, kept running with third-party patches, string and chewing gum. We'll probably pause a moment to marvel at it, as we do at old DC-3 airplanes hanging around airports.

That was a very long time ago in this industry. My first daughter, now 14, had just mastered walking and was starting to chase our poor cat around the house ("No, Annabelle, Simba's tail is *not* a handle"). PCs had no obvious rivals—smartphones and tablets didn't exist. Mobile connectivity meant Wi-Fi in the hotel lobby. Amazon was just starting to get big. Social media meant an online bulletin board. The world was a friendlier, less-dangerous place.

Microsoft tried to replace XP with Windows Vista starting in 2007, but the new Aero UI turned off a lot of people, and the need to replace hardware to run it didn't help. The Great Recession didn't do Vista

any favors, either. The User Account Control (UAC) prompts were unpopular, and still are today. (Has any user ever clicked No, even once, in the history of the universe? I call upon Microsoft to release statistics of how many exploits UAC has prevented versus its cost in users' time, as I discussed in my July 2011 column, "When Security Doesn't Make Sense," msdn.microsoft.com/magazine/hh288087.)

When the old XP hardware finally does break, its owners will naturally rethink their current needs. A full PC was the only viable option in 2001. But today, my library could get along just fine with a \$199 Chromebook for its catalog browser, using the money saved to buy more Kindle books to lend out.

Microsoft says it won't release any more security updates for XP, except for those aforementioned large, high-paying clients. I'll be curious to see how that plan holds up when the next big breach occurs. It's one thing for Microsoft not to develop updates: "Sorry, we just don't build fixes for that. We don't build them for your buggy whip, either. Here's a discount coupon for an upgrade." It's another thing entirely to develop updates and then withhold them when the next Heartbleed hits the fan: "Yep, we've got that fixed, but we're only giving it to the tax man, not you."

We'll probably keep bumping into XP now and again, kept running with third-party patches, string and chewing gum. We'll probably pause a moment to marvel at it, as we do at old DC-3 airplanes hanging around airports. (Buffalo Airways still uses them for scheduled passenger service between Yellowknife and Hay River in northern Canada. For your bucket list, perhaps?) The Second Law of Thermodynamics dictates that we'll see fewer XP systems every year.

For an XP recession, I'll give the last word to General Douglas MacArthur, whose farewell address to Congress in 1951 could describe any once-ubiquitous, now-receding technology:

"I still remember the refrain of one of the most popular barrack ballads . . . which proclaimed most proudly that 'Old soldiers never die; they just fade away.' And like the old soldier of that ballad, I now close my military career and just fade away, an old soldier who tried to do his duty as God gave him the light to see that duty. Good-bye." ■

DAVID S. PLATT teaches programming .NET at Harvard University Extension School and at companies all over the world. He's the author of 11 programming books, including "Why Software Sucks" (Addison-Wesley Professional, 2006) and "Introducing Microsoft .NET" (Microsoft Press, 2002). Microsoft named him a Software Legend in 2002. He wonders whether he should tape down two of his daughter's fingers so she learns how to count in octal. You can contact him at rollthunder.com.

facebook



Microsoft
SharePoint 2010



Linked in



twitter

SEE THE WORLD AS A DATABASE

ADO.NET ▪ JDBC ▪ ODBC ▪ SQL SSIS ▪ ODATA
MYSQL ▪ EXCEL ▪ POWERSHELL



Microsoft Visual Studio Java ODBC Microsoft SQL Server Microsoft Excel Microsoft BizTalk MySQL OData

Work With Relational Data, Not Complex APIs or Services

Whether you are a developer using ADO.NET, JDBC, OData, or MySQL, or a systems integrator working with SQL Server or Biztalk, or even an information worker familiar with ODBC or Excel – our products give you bi-directional access to live data through easy-to-use technologies that you are already familiar with. If you can connect to a database, then you will already know how to connect to Salesforce, SAP, SharePoint, Dynamics CRM, Google Apps, QuickBooks, and much more!



Give RSSBus a try today and see what mean:

visit us online at www.rssbus.com to learn more or download a free trial.

rssbus

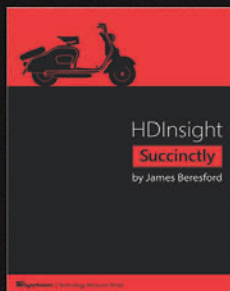
INTEGRATION YOUR WAY

Spot a green-eyed
blonde with a
star-shaped earring
on her left ear.

CAN YOU?

With good visualization and
meticulous interpretation of big data, **YOU CAN!**

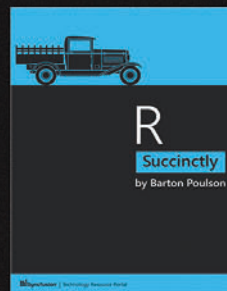
Take control of your data with free e-books from Syncfusion.



HDInsight Succinctly

by James Beresford

- Set up and manage HDInsight clusters on Azure.
- Connect with Microsoft BI tools to enrich and visualize data.



R Succinctly

by Barton Poulson

- Learn basic practices in R for analyzing your own data.
- Understand the choices that go into statistical analysis.



syncfusion.com/msdnebooks

 **Syncfusion®**